
CONFIDENTIAL

The Amiga CD32 Developer
Notes are CONFIDENTIAL.
THE INFORMATION
CONTAINED HEREIN IS
PRELIMINARY, PROPRIETARY,
AND SUBJECT TO CHANGE
WITHOUT NOTICE.

■ Copyright

Copyright 1993 Commodore-Amiga, Inc. All rights reserved. The materials used herein have been reproduced with the permission of the authors indicated. Use or reproduction of such materials shall be in accordance with the authors' instructions. Commodore and Amiga are registered trademarks of Commodore Electronics Ltd. and Commodore-Amiga, Inc. respectively. This document may also contain reference to other trademarks and registered trademarks for the various products listed which are believed to belong to the sources associated therewith.

■ Warning

The information contained herein is subject to change without notice. Commodore specifically does not make any endorsement or representation with respect to the use, results, or performance of the information (including without limitation its capabilities, appropriateness, reliability, currentness or availability).

■ Disclaimer

This information is provided "as is" without warranty of any kind, either express or implied. The entire risk as to the use of this information is assumed by the user. In no event will Commodore or its affiliated companies be liable for any damages, direct, indirect, incidental, special or consequential, resulting from any defect in the information, even if advised of the possibility of such damages.

■ Revision

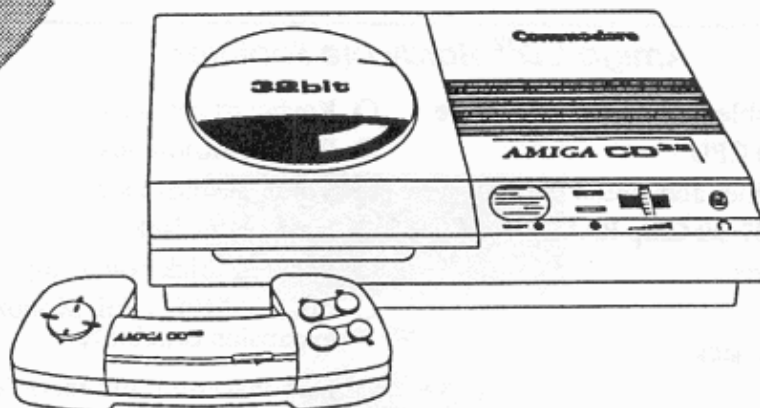
This is revision 3 of this document, August 30, 1993

Contents

Chapter:	1	Introduction	1
	2	Producing Titles	5
	3	Writing OS-Friendly Games	21
	4	Includes and Autodocs	55
	5	Expansion Connectors	93
Appendix:	A	Questions and Answers	97

1

Introduction



An exciting product has emerged from Commodore engineering. It combines the graphics excellence of the Amiga computer family with the megastorage and audio purity of the CD-ROM, making it an instant force for all to reckon with. There's a new standard in the video game world and its name is the Amiga CD³² game system.

The Amiga CD³² is an entry level, 32-bit CD game console based on the Amiga AA chip set and a 68EC020 microprocessor. The AA chips, the latest of the custom chip sets that characterize the Amiga family, provides a palette of 16 million colors, a dazzling variety of display modes and four-voice, 8-bit, stereo audio.

This is a machine that can transport players into the vortex of an alien hyperspace or the depths of a dank dungeon far below an evil warlock's castle. You'll be able to provide your starship captains with full-motion video on their viewscreens and klaxons so real, the neighbors will go to general quarters. The Amiga CD³² is a product ready to do battle in the fiercely competitive game market, but it can't go it alone, it needs a software creator. It needs you.

In this document, we present sections on the Amiga CD³² hardware and system software, title production, game programming techniques, expansion hardware, and reference documentation.

Section Outlines

- | | | |
|----------|----------------------------------|---|
| 1 | Introduction | What Amiga CD ³² is, why it exists, who will buy it and who is the competition. |
| 2 | Producing Titles | Development tools, porting existing Amiga titles, CD mastering and duplication, and distribution information. |
| 3 | Writing OS-Friendly Games | The CD ³² libraries, file system and game programming techniques. |
| 4 | Includes and Autodocs | Reference material about the Amiga CD ³² software libraries and file system. |
| 5 | Expansion Connectors | System expansion connectors and other hardware. |
| A | Questions and Answers | Answers to commonly asked questions. |

Amiga CD³² Hardware Summary

The Amiga CD³² is a game system first, then a computer. Its basic hardware configuration reflects this. The table below summarizes the hardware features.

Amiga CD³² Hardware Features

- | | |
|--|---|
| ☐ Top loading double speed CD-ROM drive | ☐ Keyboard connector |
| ☐ 14MHz 68EC020 CPU | ☐ Full expansion bus |
| ☐ Amiga AA graphics and sound chip set | ☐ Volume control switch |
| ☐ 2 Megabytes of 32-bit Chip RAM | ☐ Headphone jack |
| ☐ Two joystick ports | ☐ External brick power supply |
| ☐ S-video jack | ☐ Internal MPEG Full Motion Video (FMV) expansion capability |
| ☐ Composite video jack | ☐ Optional computer module |
| ☐ RF output jack | ☐ Multiple session disc capability |
| ☐ Stereo audio jacks | |

The Amiga CD³² Motherboard

The motherboard for the Amiga CD³² is roughly 6" by 12" and is a four layer board. On the main PCB is a 14MHz 68EC020, the AA chipset, an ASIC called Akiko, a DAC for playing standard CD audio, and a small amount of EEPROM.

Akiko is a 160-pin PQFP surface-mounted device containing most of the remainder of the logic necessary in the Amiga CD³². It includes the CD-ROM control logic and the system timers. The EEPROM is 8K bits, and designed for storing settings, high scores, bookmarks, etc., on the Amiga CD³².

In addition to the AA chips, Akiko, the CPU and the EEPROM, the main board contains 2MB of DRAM as Chip RAM. A PAL or NTSC modulator is also included on the main board.

The Compact Disc DAC is a standard 18-bit, 8x oversampling DAC. Along with the superb audio quality of the DAC, the Amiga CD³² expansion connector has digital audio input capability in order to mix an external source such as MPEG audio with the CD audio and Amiga audio.

The Amiga AA Chip Set

The Amiga AA chip set is the latest in the Amiga family of custom chips. It provides stunning graphics and animation capabilities, a huge color palette and multiple display modes. Throughout the history of the Amiga, the individual chips of each chip set have had colorful names beginning with the trio of Portia, Agnus and Daphne in the original chip set up to the latest trio of Paula, Alice and Lisa contained in the Amiga CD³² system.

- ☐ Paula controls the I/O ports, potentiometer inputs, the audio hardware and interrupt control.
- ☐ Alice is the DMA controller. It generates DMA addresses, controls Chip RAM access, video timing and includes the blitter and copper (co-processor).
- ☐ Lisa is the video data chip. It provides the 16-million color palette, playfield scrolling, sprite control, joystick and Genlock support.

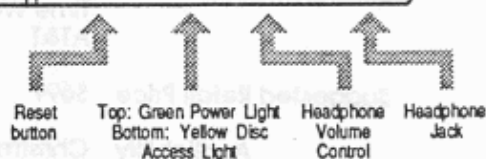
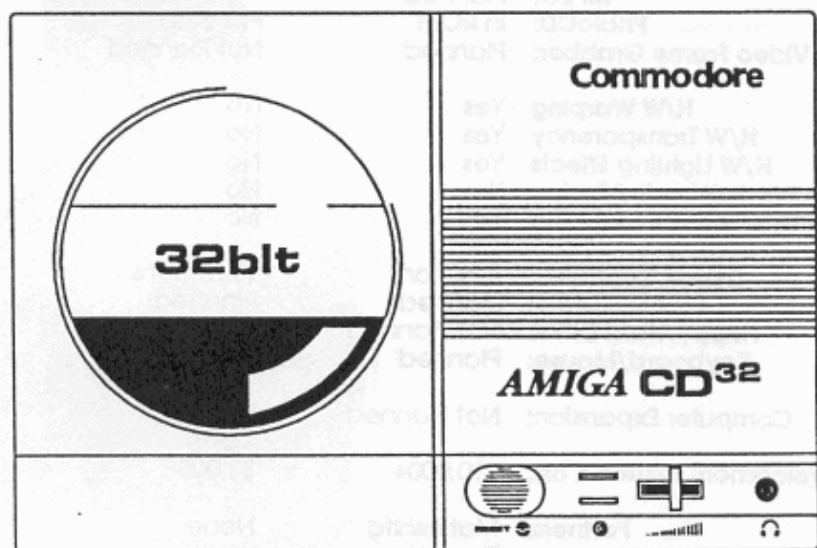
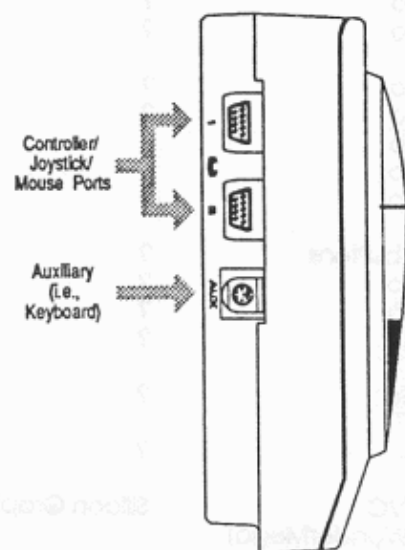
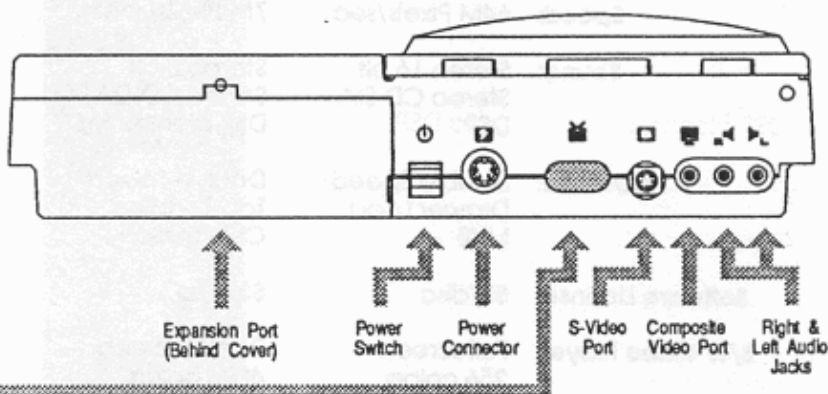
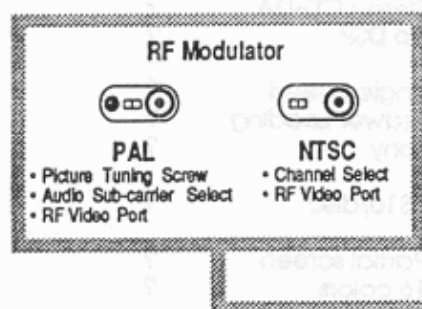
AA Chip Set Features

The AA chip set has a number of features that are ideal for games.

- 256 colors out of 16 million in all resolutions
- 32-bit wide Chip RAM and support for page mode DRAMs
- 8-bit HAM mode support (near true color display)
- Sprites are up to 64 bits wide
- Sprites can be displayed in the borders
- Attached sprites are available in all resolutions
- Dual 4-bitplane playfields
- Hardware scan-doubling support
- Hardware compatibility mode with the older Amiga ECS chip set
- Four-voice, 8 bit audio with variable sample rate

The AA chip set supports 15kHz and 31kHz display modes, but because Amiga CD³² will primarily be attached to a television set, the 31kHz modes are not used. These video modes can be used if the optional computer module is added with the RGB interface.

Controls and Ports



Amiga CD³² Comparison with other platforms

	<u>3DO</u>	<u>Amiga CD³²</u>	<u>SegaCD</u>	<u>NintendoCD</u>
CPU/Speed:	ARM60/12 mHz	68EC020/14 mHz	2x68000/12 mHz	RISC processor/?
Bits:	32 bit	32 bit	16 bit	32 bit
MIPS:	6 MIPS	3.5 MIPS	0.3 MIPS	?
2M DRAM:	1M VRAM	2M DRAM	64KB VRAM	?
Chip RAM:	1M DRAM	None	64KB SRAM	?
Fast RAM:	1KB	8KB	Yes, 7KB	?
Non Volatile RAM:				
Custom Chips:	Two animation Processors Video Processor DSP/DMA Engine Exp. Controller	I/O ports, audio and Interrupt controller DMA controller Video data controller CD-ROM controller	2 custom chips In base 3 custom chips In CD	?
Animation CELS:	Yes (100s)	8 Sprites (64 bit) & Bobs	80 Sprites (32 bit)	?
Video Modes:	640x400, 15kHz	640x400, 15kHz	320x200, 15kHz	?
Colors:	256/32768 colors	256,000/16 Million	64/512 colors	?
Speed:	64M Pixels/sec	7M Pixels/sec	??	?
Sound:	Stereo 16 bit Stereo CD-DA DSP planned	Stereo 8 bit Stereo CD-DA DSP planned	Mono 8bit FM Stereo CD-DA No DSP	?
CD-ROM:	Double Speed Drawer Load MKE	Double Speed Top Loading Chinon/Sony	Single Speed Drawer Loading Sony	?
Software License:	\$3/disc	\$3/disc	~\$10/disc	?
S/W Video Player:	Full screen 256 colors	Partial screen 4096 colors	Partial screen 16 colors	?
MPEG:	Planned	Planned	No	?
PhotoCD:	In ROM	Planned	No	?
Video Frame Grabber:	Planned	Not Planned	No	?
H/W Warping	Yes	No	No	?
H/W Transparency	Yes	No	No	?
H/W Lighting Effects	Yes	No	No	?
H/W Anti-Aliasing	Yes	No	No	?
H/W Texture Mapping	Yes	No	No	?
Game Controller:	8 buttons	11 buttons	8 buttons	?
Parallel/Serial:	Planned	Planned	No	?
Floppy/Hard Drive:	Not Planned	Planned	No	?
Keyboard/Mouse:	Planned	Planned	No	?
Computer Expansion:	Not Planned	Yes	No	?
Development System Cost:	\$10,000+	\$3,000	?	?
Partners:	Matsushita Time Warner AT&T	None	JVC (WonderMega) Pioneer	Silicon Graphics
Suggested Retail Price	\$699	\$399	\$299+\$99	\$250
Availability	Christmas goal	Sept. 1993	Now	1995

2

Producing Titles

Producing titles for the CD game machine platform is a little different than the method you might use for floppy-based titles. The basic steps are as follows:

1. Get a development system (A4000 with a hard drive).
2. Develop or port game.
3. Create an ISO image of the game with ISOCDD. Test with SimCD on an A4000.
4. Create the gold (master) disc from the ISO image.
5. Test the gold disc on a prototype CD game machine.
6. Repeat steps 2-5 until ready for duplication.

Development Systems

An Amiga 4000 with an '040 CPU is a good development platform. The A4000 has the AA chip set so the underlying hardware will be the same as the game machine. The '040 processor means that you will be able to use debugging tools such as Enforcer that require an MMU.

You should use the MapROM command with the V40 developer Kickstart release to make the A4000 more like the game machine operating system. MapROM also allows you to upgrade as new V40 Kickstarts are released (no ROMs or EPROMs required).

The IDE disk drive bundled with the A4000 gives a reasonable approximation of CD-ROM speeds. For faster, larger hard drives, use a 4091 or 2091 SCSI controller and a SCSI drive. (If you use the 2091 controller board, it must have the 7.0 ROM set to work properly with the A4000.)

Since the CD game machine has a 68EC020 processor, you may also want to have an Amiga 1200 for accurate performance testing of your product prior to mastering. The A1200 has an '020 processor and the AA chip set just like the Amiga CD³². However be aware that MMU-based debugging tools such as Enforcer will not work with the '020 processor.

The Amiga CD³² game machine is shipped with Chip RAM only. For testing, you should use either the NoFastMem command (do this early in your startup-sequence) or remove all Fast RAM from your test system so that your code loads into Chip memory only. Keep in mind that Amiga CD³² memory can be expanded so your code should be able to run with Fast memory too.

Testing with Amiga CD³² and the Debug Board

Amiga CD³² development systems are available in limited quantities that allow you to test gold (master) discs. Development systems also include a game controller and a special development plug-in card that has serial, parallel, RGB, IDE and floppy connectors. This plug-in card is officially known as the "Debug Board".

With the debug board you can run and test your code off of an IDE hard disk drive (or floppy disk). The debug board plugs into the expansion edge card connector at the rear of the game machine motherboard. The operation of the ports and their physical connectors are identical to the standard Amiga I/O ports.

Floppy drive

Supports any standard Amiga drive (A1010, A1011). Do not plug or unplug drives from this port with the power on. The 8520s connected to this port are very static sensitive and can be easily damaged this way. This port has the highest boot priority.

Parallel port

Identical to other Amiga parallel ports.

Serial port

Identical to other Amiga serial ports except that Ring Indicate is not supported. This is useful for serial port debugging with the debug.library and other tools such as Enforcer.

IDE Interface

This can support any size IDE hard drive with a 40-pin interface connector. A separate power supply must be provided for the hard drive. The connector is not keyed so carefully note the pin 1 orientation. This interface is memory-mapped I/O, not DMA, so it is relatively slow. The hardware implementation on the debug board is even slower than an A1200 IDE interface so you may notice some exceptionally long boot times from hard disk.

Emulator

The socket on the debug board labelled U1 allows you to connect a 68020 In-Circuit-Emulator pod to the system. The 68020 ICEs can be used in place of the 68EC020 without any problems. To enable the emulator, short the two pads of XJ1 on the debug board together. (This asserts a bus request signal to the 68EC020 processor on the main board disabling it.)

Patch RAM

This optional RAM may or may not be installed on your board (depending on its vintage). This is 512K bytes of static RAM located at \$F0 0000 used for patching the operating system firmware. It is not needed for normal debugging operation.

RGB video

This is analog RGB. The connector may or may not be present on your board (depending on its vintage) but the pinout is identical to Amiga video, so a cable could be hard-wired to the board. If this is too much trouble, use the S-video signal from the main PCB; it is nearly as good as the analog RGB.

Note that if you have one of the early prototype systems, the debug board is required in order for the system to operate. The prototype systems cannot boot without the 8520 CIAs that are part of the debug board. Installation of the debug board should be done with caution. It requires a good deal of force to insert the board and the pins inside the connector are fragile. Once bent, the pins cannot be straightened.

Development Tools: ISOCD, SimCD and OPTCD

There are three development tools that can help you in your effort to make CD titles:

- ISOCD** - Creates an ISO-9660 image file suitable for making the gold (master) disc. Prepares a partition for use with SimCD.
- SimCD** - Emulates the CD-ROM drive (CD0:) from a partition prepared with ISOCD. This allows you to test the integrity of your ISO image without having to cut a gold (master) disc.
- OPTCD** - Optimizes the arrangement of data on a CD to minimize seek times and improve performance.

ISOCD, SimCD and OPTCD are all part of the ISO Tools Disk available to any licensed Amiga CD³² developer. With ISOCD, SimCD and OPTCD you can create and test the integrity of ISO images on hard disk before taking the time and expense to cut a gold (master) disc.

ISOCD has two purposes. It prepares a partition for use with SimCD and it also can produce the ISO-9660 image file you need in order to create the gold disc. The basic steps to using ISOCD are as follows:

1. Set up a partition of the appropriate size on hard disk named CDN:. (Do not call it CD0:. That name is reserved.)
2. Run ISOCD. Click on the Source gadget to specify the source directory containing your program and data files.
3. Click on the Image gadget to specify the target partition to use for the ISO image. Only specify a file name for the image if you are ready to cut a gold (master) disc.
4. Click on the Examine gadget. A list of files is displayed. (Do not use the "Add Empty Space" feature with V1.00; it is broken. This option works in 1.03 and later.)
5. Click on the Build gadget. Exit.

SimCD sets up a simulated CD-ROM drive named CD0: using the ISO image set up previously with ISOCD. With SimCD, you can check to see if the ISO-9660 version of your application will work. There is no need to cut a disc. The steps to using SimCD are:

1. Make a directory called devs:Local. Copy devs/local/cdtv.device from the ISO Tools disk into devs:Local.
2. Make a directory called l:Local. Copy l/local/cdfs from the ISO Tools disk into l:Local.
3. Make 2 assignments. Assign devs: devs:Local add. Assign l: l:Local add.
4. Load SimCD. Under Configuration, select CDFS and cdtv.device only.
5. Set "Sim Device" to CDN: and hit the Simulate button.
6. SimCD sets up a simulated CD0: drive using the ISO image on CDN:. Move to the directory CD0: and run your startup-sequence or executable.

**Warning:**

SimCD emulates the older cdtv.device driver used in Commodore CDTV systems. Commodore is working to upgrade this important tool so that it emulates the new cd.device driver used in the CD game machine. Until then, be aware that SimCD cannot be used to test code that directly calls the cd.device or uses CDXL reads. Also, SimCD emulates the slower speeds of the CD-ROM drive used with CDTV. Your title on disc will be faster.

Once you have tested your application in its ISO-9660 version with ISOCD and SimCD, you are ready to create the gold (master) disc.

▶ Trademark File

In order to get a title to boot on an Amiga CD³² console, a special Amiga CD³² trademark file must be included directly after the PVD sectors. CDTV used a similar trademark file either right after the PVD sectors or in a normal file in the root of the CD directory structure. This is not the case with AmigaCD³², the trademark file *must* always be included after the PVD sectors.

The Amiga CD³² trademark file differs from the CDTV one. The Amiga CD³² system relies on the differences in the trademark files to determine if a title is a CDTV title or an Amiga CD³² title. Contact CATS for information on obtaining the trademark file if you plan on generating your own ISO image. In order to ensure the trademark file gets placed directly after the PVD sectors, do not simply copy it to your CD's directory. Instead make sure that the trademark file is in the directory you are currently CD'd to (probably where ISOCD is located) when creating your ISO image.

Note that use of the trademark file requires an Amiga CD³² license agreement with Commodore. Contact CATS for licensing information.

The Amiga CD³² system is compatible with some older CDTV titles. The type of trademark file on a CD determines the environment in which the title is started. If a CDTV trademark file is found, the title is booted in ECS mode. If an Amiga CD³² trademark is found, the title is booted directly in AA mode. In addition, many system patches are applied on a per-title basis when booting CDTV titles in order to keep them working on the Amiga CD³².

▶ Creating the Gold Disc

CD manufacturing companies make a gold (master) disc before they produce your title in large quantities. This step is known as pre-mastering and can usually be performed by the same company that produces the discs in quantity.

For a limited time, Commodore will help you create the gold disc. This service is provided to shorten the amount of time it takes to bring your product to market. (It is OK to use a pre-mastering service of your own choosing if you prefer.) To take advantage of this service, send the ISO image of your application on DAT tape to the Maidenhead office in the U.K., care of David Pocock. A gold disc will be created and, if it works, an additional disc will be created for examination by Commodore, West Chester. In the U.S., send the DAT tape directly to the West Chester office care of Larry Feldman.

The gold disc will be returned to you as quickly as possible so that you can test it on a CD game machine prior to manufacturing it in quantity. When the gold disc is created, any protection bits or script bits you have set for your files will be lost since these are not supported by the ISO-9660 standard. Be sure to avoid counting on the presence of these bits in application code.

There are a variety of ways to convey your ISO image to the pre-mastering service. Depending on the vendor, you might send off a hard disk, a DAT tape containing a copy of the ISO, 1/2" U-Matic video tape — some vendors even accept floppies. If you use Commodore for your pre-mastering service, try to provide your ISO image on DAT tape, if possible. Other formats will take longer. If you do not supply an ISO image on DAT tape, be sure to include the volume name you want to use for the CD-ROM disc. If your title also includes CD audio, you should try to provide this on separate media, preferably a DAT tape.

■ Manufacturing CD-ROMs

While it is possible to create a "master-stamper" directly from the gold (master) disc, most replication centers prefer to receive a 9-track tape. This master tape is an ANSI-labelled, 9-track, 1/2" tape containing the ISO-9660 image of your code and data. This can be prepared by you or by the service that created the gold disc.

The CD manufacturer will transfer the tape to a laser beam recording system which creates the master stamper. The stamper presses CDs from molten polycarbonate which is subsequently coated with aluminum and a protectant. Labels are added and the disc inserted into a jewel case or other packaging.

In the U.K., the recommended vendors for volume production of your CD-ROM based application are:

SonoPress

26/27 Conduit St.
London W1R 9TA
Voice: 71 499 6813
Fax: 71 493 7244

Nimbus

Wyastone Leys
Monmouth
Gwent NP5 3SR
Voice: 600 890 682
FAX: 600 890 779

Pricing is subject to negotiation and volume but a rough rule of thumb is:

Mastering: £ 900
Set up: £ 200
Per disc: £ 0.90 to 1.50

There are many other vendors you can choose to work with. Here is a brief list to get you started in your search:

3M Optical Recording Department

3M Center 223-55-01
St. Paul, MN 55144-1000
Telephone (612) 736-5399
Fax (612) 733-0158
Sales contact: Don Winklepleck, (603) 595 0391
Technical contact: Andy Axelsen, (800) 336 3636
Fax: (715) 235 0500
Territories served: North and South America, Europe, Far East
Preferred source format: 8mm Exabyte, 4 mm DAT, 9 Track, MO, Hard Disk, CDWO, 3480, DC6150 (QIC)
One-off service available: Call
Terms of business: Net 30 days
Pre-mastering and CDTV ISO building: Call

Disc Packaging and Graphics Standards

Before your title goes into production, you will need to determine its packaging. Will your discs be packaged in jewel cases? Do you plan to have any outer packaging for your disc cases? What about additional documentation, reference cards and promotional materials? All of this should be determined before hand, since most duplication houses requires packaging or packaging specs prior to or with a disc production order. Some disc duplicators provide design and printing services, relieving you of this burden.

The packaging decision is yours, all that Commodore asks is that you try to adhere to its graphics standards. This section provides the information needed to present a unified image to the public. A universal standard for the visual identity of all Amiga CD³² products.

The Logos

The Amiga CD³² logo is the foundation for the entire identity program. It consists of the of the Amiga CD³² logo type and the "TM" symbol. These elements should *never* be altered from the form shown below, except as specified in this publication. The logo's attributes are as follows:

- The Amiga CD³² logo must appear on a 100% black background.
- "Amiga" and "32" and the "TM" symbol print in PMS red 485 or equivalent. If the logo is to appear in only black & white, the previous words should be printed in a 20% black screen.
- "CD" is a white knockout (or prints in white)
- In most cases, the logo should be positioned horizontal in the upper left corner of the package.



In most instances the Commodore logo should appear in conjunction with the main Amiga CD³² logo. When this is the case, the Commodore logo should have the following attributes:

- "Commodore" is a white knockout (or prints in white)
- It is always positioned to the left of Amiga CD³², in the upper right corner of the package.
- The top of the logo lines up with the top of the "32" in the Amiga CD³² logo.
- The minimum black space between the two logos is one full length of the Commodore logo.



ALWAYS USE PHOTOMECHANICAL REPRODUCTIONS OF THE LOGOS.

NEVER...

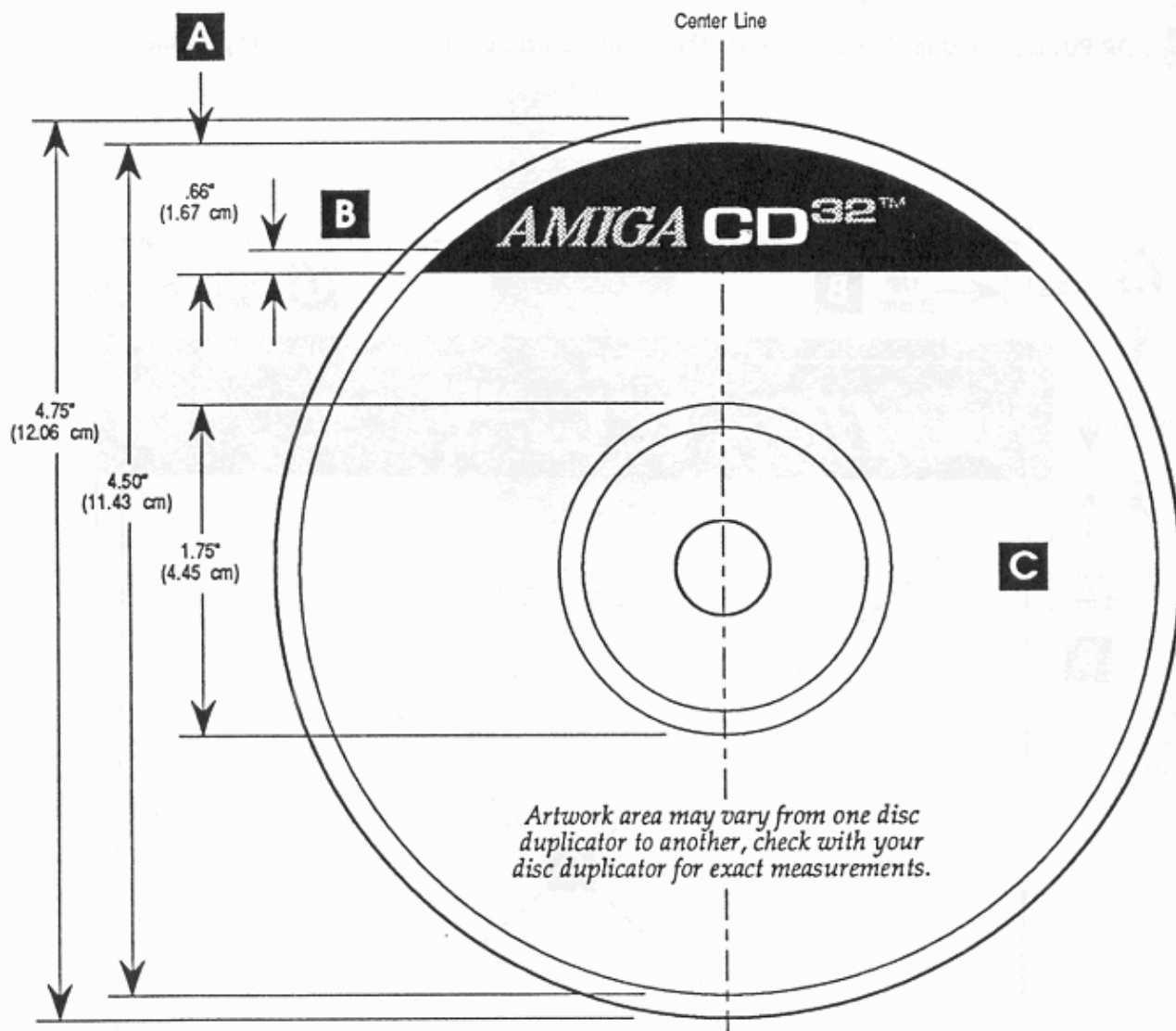
- Use photocopies of the logos.
- Attempt to redraw or re-create the logos yourself.
- Reproduce the logos from the diagrams or templates in this manual.

Reproducible "stats" can be acquired from the CATS department at Commodore. Logos are also available in Encapsulated Postscript format for use in creating packaging on desktop publishing computer systems.

Disc Graphics

Below is the proper application of graphics on the face of the disc itself. The only specification is the consistent application of the Amiga CD³² logo over a black bar.

- A** TOP BAR: A band of 100% black, .66" tall to the edge of the disc's graphic area.
- B** AMIGA CD³² LOGO: Centered horizontally within the top bar.
- C** FOR PUBLISHER USE: The remaining 4 1/2" DIA. area is free to use for title graphics



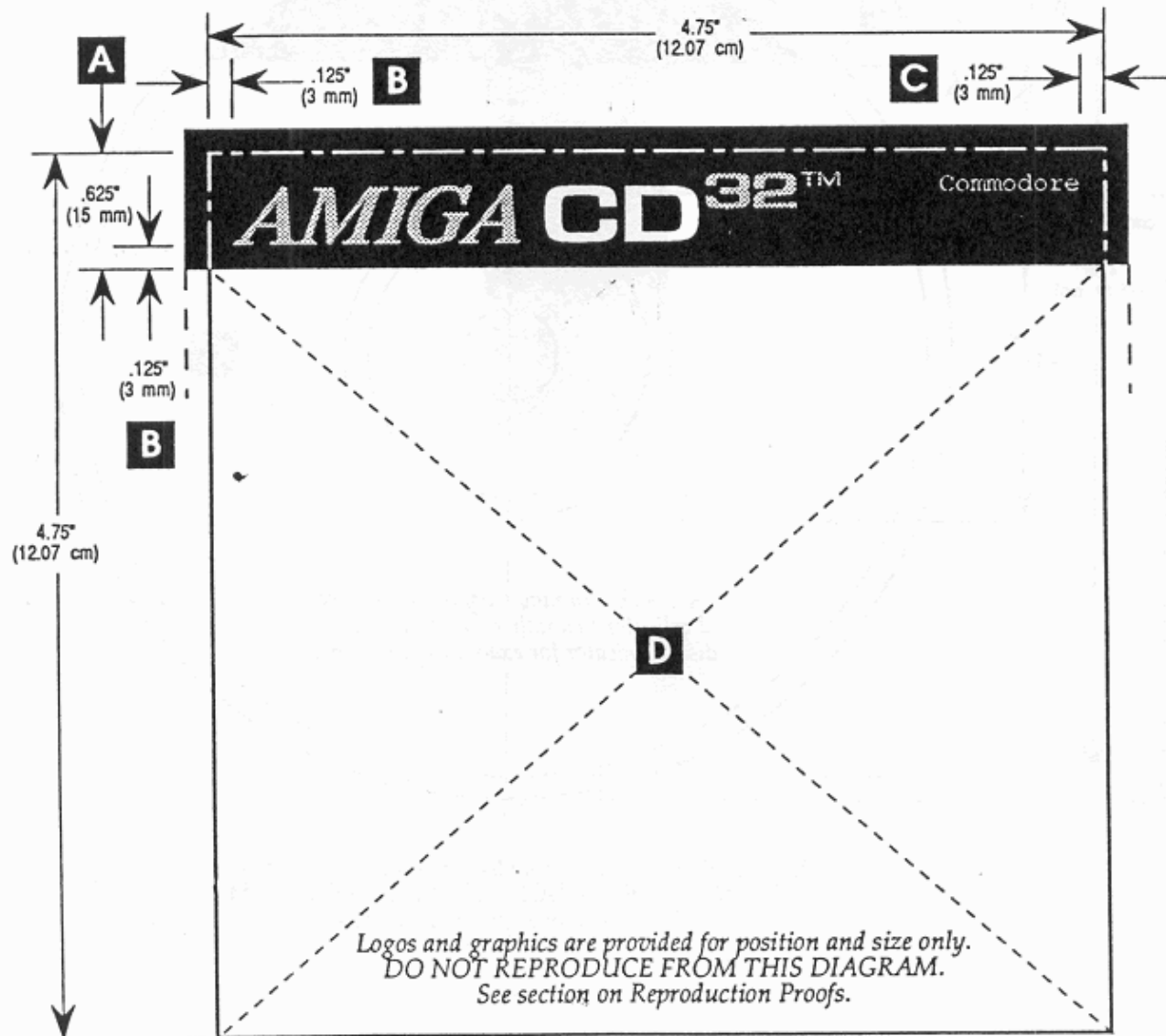
Logos and graphics are provided for position and size only.
DO NOT REPRODUCE FROM THIS DIAGRAM.
See section on Reproduction Proofs.

Cover Insert

The cover insert is either a single sheet or a multipage booklet placed inside the jewel case cover. For the purpose of these standards, Commodore is only concerned with the placement of its graphics on the front cover portion of the insert.

Before designing the cover insert and choosing paper stock, check with your disc duplicator for exact measurements and paper requirements. This will avoid jamming or damage during insertion.

- A TOP BAR:** A band of 100% black, 5/8" tall should bleed off the top and both sides of the card.
- B AMIGA CD³² LOGO:** Positioned as shown.
- C COMMODORE LOGO:** positioned so that the top lines up with the top the Amiga CD³² logo.
- D FOR PUBLISHER USE:** The area within the dashed borders is free to use for title graphics.



► Inlay Card Graphics

The inlay card is a flat sheet of paper which is scored and folded approximately 1/4" from both the left and right edges. When encased inside a jewel case, it becomes the graphics for the back and side panels. For a double CD jewel case, an identical inlay card is required for front and spine artwork.

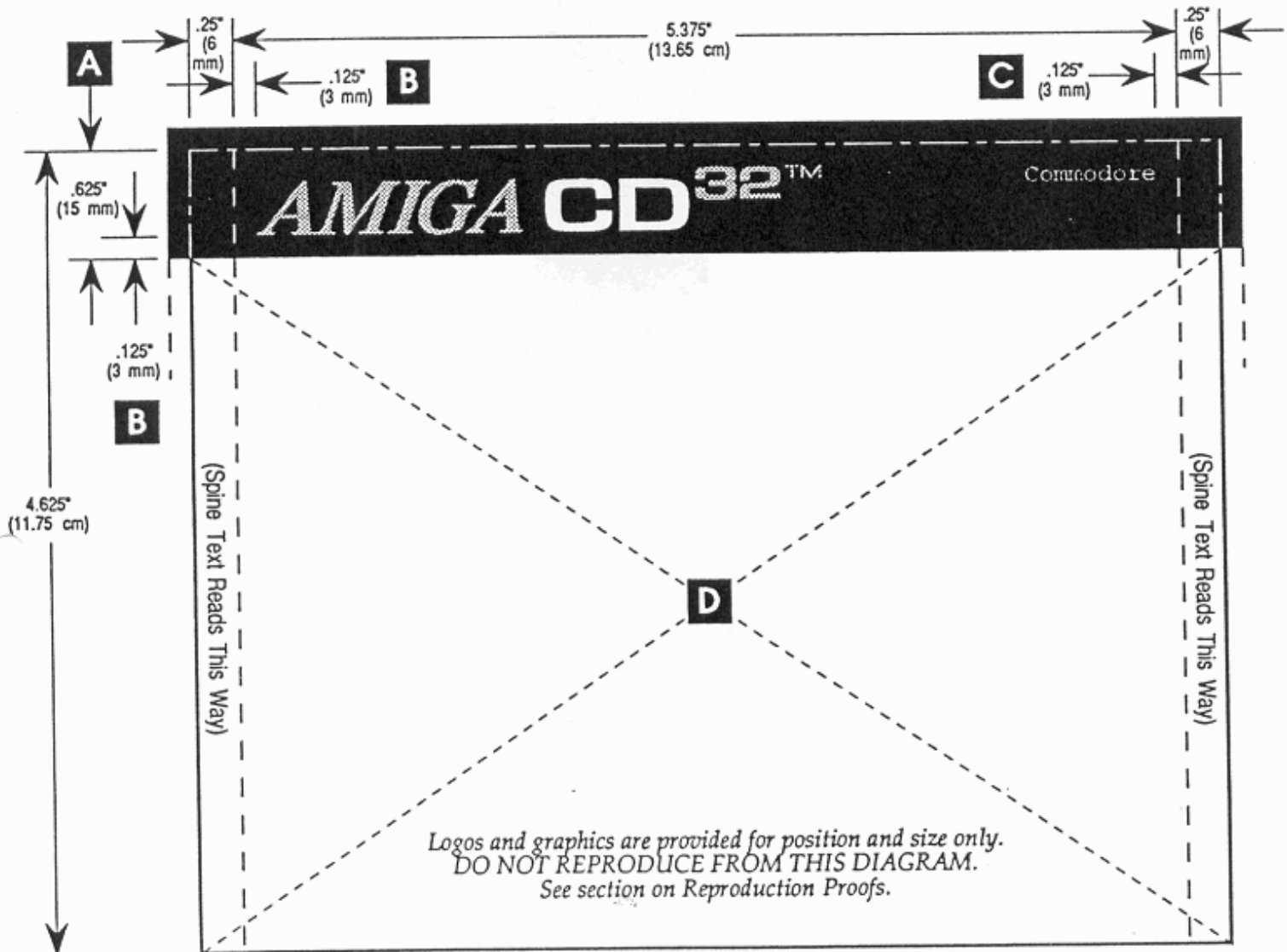
Before designing the inlay card and choosing paper stock, check with your disc duplicator for exact measurements and paper requirements. This will avoid jamming or damage during insertion.

A TOP BAR: A band of 100% black, 5/8" tall should bleed off the top and both sides of the card.

B AMIGA CD³² LOGO: Positioned as shown.

C COMMODORE LOGO: positioned so that the top lines up with the top of the Amiga CD³² logo.

D FOR PUBLISHER USE: The area within the dashed borders is free to use for title graphics.



3

Writing OS-Friendly Games

Many Amiga programmers believe that the only way to write a whiz-bang, speed-of-light game is to bypass the operating system and go straight to the hardware. A better approach is to write games that are OS-friendly. The combination of the AA chipset and the latest V40 system software make system-friendly solutions more possible than ever.

► Why Use the Operating System?

Here are some reasons to use the OS for games:

- OS code runs out of ROM, which is faster than running equivalent code out of Chip RAM.
- Why reinvent the wheel? Spend your time doing things that only you can do.
- The OS automatically supports pre-ECS, ECS, and AA (and will support the next revision of the Amiga chip set).
- There is less code to write. The OS has routines for handling all screen positions and scrolls, mouse movement, etc. This means less development time and less time getting the hardware to work, and more time making the game more playable.
- More robustness. For instance, the OS floppy disk code is far less picky about drive parameters than 99% of custom floppy I/O code.
- Hides bugs and quirks of the chip set. The AA chip set has a few bugs which the OS hides from you.
- Multiple platforms. OS code will run on all Amiga-based machines, whatever their flavor.
- Compatibility with future chipsets. The next revision of the Amiga chip set will not be register-level compatible with AA.

► Things the System Cannot Do

A long-term goal of the Amiga operating system is to support the common display tricks often used in games. There are some things that can't be done though:

- Scrolling individual scan lines of a ViewPort
- Using a copper list to fade a AA color register
- Dynamic updating of user copper lists.

All these are planned to be addressed in future OS releases. The goal is to allow these and other types of operations within normal Intuition screens if possible.

► ECS-to-AA Incompatibilities that the OS Handles

If you are coding for the AA chip set (and you ought to be) the OS handles some problems of backward compatibility with the older ECS chip set that you do not want to deal with.

- Vertical counter behaves differently in programmable beam modes.
- No SuperHires color scrambling.
- Bitplane alignment problems.

► Compatibility Beyond AA that the OS Handles

The successor to the AA chip set will not be register-level compatible. To minimize the side-effects, the operating system will have features designed to hide any incompatibilities from programs that use system calls. Hence, using the operating system for display operations is the best way to ensure a measure of forward-compatibility with chip sets to come.

Some of the problems the system will handle for you are:

- Future chip sets will have no fetch-mode selections. All selections will be automatic.
- Different DDFSTART/STOP calculations for single/dual systems.
- Color loading will be different
- Exact copper timings will be different
- No SuperHires
- Multiple blitters

► Booting A Title

Amiga CD³² boots just like a normal Amiga. The only available boot device is normally the CD (CD0:), so the system boots off the CD in the same manner it would boot from a hard-drive, executing a startup-sequence and so on. In order to boot, your title must have the Commodore trademark file right after the PVD sectors (see chapter 2 for more on this.)

Competing game consoles generally have instantaneous startup times for titles, due to the cartridge interface used. Amiga CD³² benefits in many ways from its CD-ROM technology, however booting time is not one of these benefits. Due to the very slow seek times of CD-ROM devices, the dynamics of disk I/O are quite a bit different from a hard drive. It is crucial that developers be aware of this situation and take appropriate action to reduce boot times.

The startup animation sequence provided by the Amiga CD³² ROM has some continuous motion to indicate that the system is doing something while a title is booting. This animation sequence requires little memory and gets out of the way as soon as the title is ready to run. This animation helps eliminate the long periods of black screen familiar to CDTV users.

An important factor in bootup time is the length of the startup-sequence executed by the system prior to running a title. The startup-sequence shipped with standard Amiga systems is intended for use in a multitasking computer environment. Understanding the startup-sequence, and optimizing it can help reduce bootup time considerably.

The following is a startup-sequence which is adequate for most titles. It is much smaller and many times faster than the standard startup-sequence provided with Workbench 3.1. It also leaves noticeably more memory for your title than when the standard startup-sequence is used.

```
C:SetPatch QUIET

; Only include this line if you are using locale.library
C:Assign >NIL: LOCALE: SYS:Locale

; Here you can add any assignments that are specific to your title.
C:Assign >NIL: WestChesterWarrior: SYS:WestChesterWarrior

; Only include these lines if you include the corresponding monitor files on
; your disc. Only include the files on disc if you actually use the mode these
; monitors provide. Each monitor included eats up memory and takes time to load.
DEVS:Monitors/DblNTSC
DEVS:Monitors/DblPAL

; Here you just start your title
SYS:WestChesterWarrior
```

Note that the use of *any* Workbench files on a CD (other than plain text files such as the startup-sequence) requires a Workbench License Agreement with Commodore. Contact CATS for licensing information.

Using explicit path specifications as shown above (C:Assign instead of just Assign) avoids a lot of head seeking and can improve startup performance. Another important consideration in bootup time is the size and complexity of the title itself. It is highly desirable to get your title on the screen as soon as possible. This can become difficult with very large titles which take a long time to load. The solution to this problem involves either the use of overlays, or the use of a separate loader program.

The simpler approach is the loader program. This is a very small program that gets run in the startup-sequence (just like WestChesterWarrior in the example above). This loader program contains some animation and music sequences that can quickly be put onto the screen. While this animation is playing, the loader program can then proceed to load the rest of the title into memory and activate it. See the DOS RunCommand() or System() functions for information on how the loader program can invoke the main title.

Applications designed to run on other Amiga systems (in addition to Amiga CD³²) might require a more elaborate startup-sequence. In that case be aware that references to DF0: in the startup-sequence will cause a boot failure on an Amiga CD³² system. Similarly, a startup-sequence that executes other scripts must have T: assigned to RAM: in order to work properly on Amiga CD³² systems.

Startup Environment

While the system is loading a title into memory, there is a small animation playing on the screen. This animation tells the user that the system is busy and will be done shortly. This animation consumes little memory, and exits the system as soon as the title indicates it is ready to run. Your application can control this animation with a special Amiga CD³² library, the freeanim.library. Here are some things to keep in mind for dealing successfully with the startup animation:

- The startup animation runs in an Intuition screen. In order for the exit sequence of the animation to work correctly, titles must not take over the display prior to the animation shutting down.

- The startup animation currently does not use the audio.device but this could change in future versions. It is therefore important for titles to avoid playing any music prior to the animation shutting down unless they first ensure that all channels on the audio.device are available (i.e., not in use).
- When the animation exits the system, it leaves the display completely black. You can assume this in your titles, and fade up from black, or whatever other transitions you wish to do.

Once your title is in memory and wishes to display something on the screen, it needs to do:

```
FreeAnimBase = OpenLibrary("freeanim.library", 0);
```

This will tell the startup animation to begin shutting down, which involves removing itself from the display and freeing the resources it uses. This may take a while, up to about a second. While the startup animation is shutting down, your application can prepare data to display and generally initialize itself. This parallelism reduces the time the user has to wait for the titles to start.

Once you have done what you need to in order to display your title screen, you should do the following:

```
CloseLibrary(FreeAnimBase);
```

This will wait for the animation to complete its shutdown. Once this function has returned, it is now safe to display your title screen. If you have no need to process information while the animation is shutting down, the following will work:

```
CloseLibrary(OpenLibrary("freeanim.library", 0));
```

It is important to note that users may try to run your code on a regular Amiga computer equipped with CD-ROM, i.e. not an AmigaCD³². In that case, there will not be a freeanim.library around, so your code should not fail when freeanim.library cannot be opened. This is easy to do, just don't put any error checking in:

```
FreeAnimBase = OpenLibrary("freeanim.library", 0);
/* initialize application stuff */
CloseLibrary(FreeAnimBase);
/* start application animation/display */
```

CloseLibrary() accepts a NULL pointer and ignores it. Therefore if OpenLibrary() fails and returns NULL, CloseLibrary() will still work.

▶ Running Your Titles From Workbench

If Commodore decides to make Amiga CD³²-compatible hardware available for its other Amiga computer systems, it would be highly desirable if any existing Amiga CD³² titles could be run from Workbench on those systems. This requires only a few lines of code to support:

1. You should handle disc insertion and removal events. On a computer system, it would not be wise to enable the "reboot on removal" feature of cd.device. When running on an Amiga CD³², you should generally reboot when a disc is removed, but in a computer environment, you should deal with this eventuality and handle it gracefully by putting up an error message, or at least quitting the title. Crashing is never a good idea.

2. Your code should handle the Workbench startup environment.
3. Your CD should include icons for the appropriate files.
4. Your code should be able to run without having the startup-sequence from your CD run. This is important. If your program relies on assignments made in the startup-sequence, these will not be present when running from Workbench. An easy solution is to provide an IconX script on the CD that gets run from Workbench and loads in the title. The script can do the needed assigns.
5. Your code should multitask nicely. It should release all its resources on exit. If this is not possible, you must warn the user in the documentation and in the software itself that starting this title will take over the system and cause any unsaved work to be lost.

The startup environment in a computer system will be different from the normal Amiga CD³² environment. The `freeanim.library` will not be there, so your code should not fail if `freeanim.library` cannot be opened. In addition, there will not be a startup animation in the system, so the initial display will not be a black screen and will instead be the user's Workbench screen.

It would also be a nice touch to make the software runnable when not on the CD. This would allow users to copy the title to their hard drive, enabling potentially much faster operation.

Finally, if you wish your application to look right at home on a CD-ROM-equipped Amiga computer, you should avoid direct references to "Amiga CD³²" within your title. The title might actually be running on an A1200 or A4000.

■ Preferences

CDs should not contain any preferences files. This specifically means that the following files should not be on your CDs:

```
Devs/system-configuration
Prefs/env-archive/sys/#?
```

Including any preferences file can cause odd behavior during the startup animation sequence.

In addition, many CDTV titles used to call the `Intuition SetPrefs()` function in order to change screen positioning and centering. This is not the correct method and will lead to problems with the startup animation. See the section below on screen positioning for information on how to do it right.

■ Using System Graphics

There are three basic ways you can set up a display on the Amiga:

- Intuition library `OpenScreen()`
- Graphics library `MakeVPort()`, `MrgCop()`, `LoadView()`
- Hitting the hardware registers directly

Using Intuition to set up the display is by far the easiest but also has the greatest memory and processing overhead. A reasonable middle course is to set up the display using the lower-level functions provided in the graphics library. In that case, you do not have to coexist peacefully with Intuition (nor do you get any of the nice services it provides). A detailed example showing how to set up your own display with the `graphics.library` is listed at the end of this chapter.

The worst method of setting up the display is by hitting the hardware registers directly. As the Amiga chip set improves, so many of the internal graphic mechanisms will change that applications using direct access techniques really have no chance of compatibility with future machines.

Graphic Displays

The typical user of the game machine will be using it on the family television set. As such, the display mode that makes the most sense is a low-resolution, 320 x 200 (NTSC) or 256 (PAL), 8-bitplane, 256-color screen. Most games today use the entire display area (i.e., bezel to bezel). To do this on the Amiga requires the use of overscan (more on this in the next section).

The game machine has Chip RAM only, hence, memory is shared between the custom graphics chips and the CPU. Contention between the custom chips and the CPU is an issue to consider. A low-resolution, 8-bitplane display does not involve any cycle stealing from the CPU assuming that the OS has been left in place so that the higher fetch-bandwidth modes of the system are in effect.

Screen Positioning with the OS

With overscan, the Amiga's nominal display dimensions can be extended so that graphics are displayable in what is normally the border area. Prior to Release 2 of the operating system, there was no officially supported way to control overscan displays and only limited support for screen positioning control. Numerous ad hoc techniques of varying quality were invented, and most are still supported, though some only work because of patches designed to keep them operational.

Release 2 (V37) introduced the graphics database and a host of structures to help you control your screen position. For correct operation on the Amiga CD³² console, we strongly urge you to use these functions and structures. First, some definitions to help you along:

Text Overscan: this is the old "morerows" size from 1.3, and has been called the "Text Overscan" or "Text Size" in V37+ Overscan Prefs. This rectangle defaults to a nominal-sized rectangle, but can be enlarged and/or repositioned by the user via Overscan Prefs. If this rectangle is correctly set, no pixel in Text Overscan is supposed to be masked by the bezel of the monitor.

Display Clip: a rectangle that describes the desired displayable portion of your screen. The coordinates are in screen-pixels, and the origin is the upper-left corner of the Text Overscan rectangle.

It is best to center your own display with respect to the Text Overscan rectangle. Here's a piece of code to handle it:

```
/* CenterDClip( displayID, width, height, rect )
 * Fills in the struct Rectangle pointed to by rect with a DisplayClip
 * whose size is the width and height specified, and whose position
 * is centered around the Text Overscan rectangle for the specified
 * displayID.
 *
 * Returns TRUE if it succeeds, FALSE if it doesn't (usually caused
 * by an invalid displayID).
 */
BOOL CenterDClip(ULONG displayID, ULONG width, ULONG height,
                 struct Rectangle *rect )
```

```

{
    BOOL success = FALSE;
    LONG excess_x;
    LONG excess_y;

    if ( QueryOverscan( displayID, rect, OSCAN_TEXT ) )
    {
        excess_x = ( width - ( rect->MaxX - rect->MinX + 1 ) );
        excess_y = ( height - ( rect->MaxY - rect->MinY + 1 ) );
        rect->MinX -= excess_x / 2;
        rect->MaxX = rect->MinX + width - 1;
        rect->MinY -= excess_y / 2;
        rect->MaxY = rect->MinY + height - 1;

        success = TRUE;
    }
    return( success );
}

```

Sample use:

```

#define MY_DISPLAY_ID HIRES_KEY
#define MY_WIDTH      whatever
#define MY_HEIGHT     whatever
...

struct Rectangle dclip;

if ( CenterDClip( MY_DISPLAY_ID, MY_WIDTH, MY_HEIGHT, &dclip ) )
{
    if ( myscreen = OpenScreenTags( NULL,
    SA_DClip, &dclip,
    ...
    TAG_DONE ) )
    }

```

Centering yourself using CenterDClip() could theoretically give you a Display Clip which extends outside of the OSCAN_MAX Display Clip, which represents the hardware limits. If you would rather be slightly off-center than clipped, you may wish to shift the resulting DClip to fit within Maximum Overscan. The easiest way to get that rectangle is:

```
QueryOverscan( displayID, &maxrect, OSCAN_MAX );
```

► New V39 and V40 Graphics Capabilities

The graphics.library for V39 gained many features that make creating system friendly, fast moving graphics and animations much easier. Autodocs for the graphics.library are a good source of information on these new features. Some examples are also available from the 1993 DevCon notes. Of specific interest is the new double-buffering support which is even available in Intuition screens.

V40 of graphics.library which is present in the Amiga CD³² ROM contains a few extra features specifically targeted at improving the performance of animation and rendering code. These features are relatively new and not well known, so they are briefly explained below.

ViewPorts that don't load colors. ViewPorts normally always have a copper list associated with them which loads up all the colors needed for their depth. This new feature makes it so that only color 0 gets loaded within the copper list. For deep screens, this will substantially reduce the copper list length. The rest of the colors are inherited from the viewport above. This feature is useful to reduce the inter-Viewport gap, and for faster color cycling. See the VC_NoColorPaletteLoad tag in <graphics/videocontrol.h> and in the graphics.library/VideoControl() Autodoc entry.

Preventing intermediate copper list updating. This speeds up LoadRGB(), SetRGB(), ScrollVPort(), and ChangeVPortBitMap(). See the VC_IntermediateCLUpdate tag in <graphics/videocontrol.h> and in the graphics.library/VideoControl() Autodoc entry.

Dual-playfield disable. Disables setting of the dual playfield bit in dual-playfield ViewPorts. Odd and even bit-planes will still scroll separately, but will be fed directly to the palette. This can be used for instance to do cross-fades between independently scrolling images. See the VC_DUALPF_Disable tag in <graphics/videocontrol.h> and the graphics.library/VideoControl() Autodoc entry.

Amiga CD³² includes very fast chunky-to-planar conversion hardware. This allows for high-quality 3D graphics. Consult the documentation for the WriteChunkyPixels() function in the graphics.library Autodoc file for more information. [Note: this function is not yet in the FD or protos but there is a pragma in include/pragmas/graphics_pragmas.h]

Note that the above new ViewPort features work fine in Intuition screens as well. You just need to pass the screen's ViewPort as a parameter to the VideoControl() function.

You might also wish to look at the documentation for the SA_VideoControl and SA_MinimizeISG tags for the OpenScreenTagList() function. These two tags offer Intuition-level control of the new features outlined above. There are also the SA_Interleaved, SA_Exclusive, and SA_Draggable tags which can greatly help writing animation and graphics code. All these tags are documented in <intuition/screens.h> and in the intuition.library Autodoc.

V40 also added many new display modes, four of which are especially useful for animation support. These modes are defined in <graphics/modeid.h> as:

```
/* Added for V40 - may be useful modes for some games or animations. */
#define LORESSDBL_KEY          0x00000008
#define LORESHAMSDBL_KEY      0x00000808
#define LORESEHBSDBL_KEY      0x00000088
#define HIRESHAMSDBL_KEY      0x00008808
```

These new V40 display modes allow you to display a given raster at twice its normal height. Each line of the raster automatically gets repeated on the screen as it is displayed. This allows a 128 pixel high raster to be displayed as if it were 256 pixels tall. This can be very helpful in order to create full screen animations. For example, a 128 pixel high CDXL animation can suddenly occupy the full screen.

When you use one of these new modes, you *do not* allocate more raster or open a taller screen, i.e., you still allocate a raster of the smaller height (128 for example), and if you are blitting the image you still only blit 128 lines. Doubling the height of your image is a video effect accomplished by the AA hardware. These modes work on a composite monitor or TV.

For more on how to set up a display using graphics.library, see the example at the end of this chapter.

► Display Mode Promotion

Mode promotion is a feature which converts 15kHz screens into 31kHz screens, in order to remove interlace flicker, or inter-line gaps. It is the AA equivalent of the A3000 display enhancer hardware.

Mode promotion can be turned on or off by the user. When turned on, screens opened using the 1.3 OpenScreen() function, or opened using a default monitor ID will end up promoted to a 31kHz mode. The promotion process is not totally transparent as certain aspects of the copper list, the refresh rate, and available overscan dimensions are different for the promoted state than for the unpromoted state. These differences can be crucial for many developers.

To avoid promotion affecting your titles, you simply need to open your screens with explicit mode IDs using the `PAL_MONITOR_ID` and `NTSC_MONITOR_ID`, instead of using `DEFAULT_MONITOR_ID`.

Note that promotion is not turned on in Amiga CD³² systems, the problem can only surface in a computer environment.

Consult the ROM Kernel manuals, the 1991 and 1993 DevCon notes for more information on display mode IDs and display mode promotion.

Intuition Screen Defaults

Applications that use Intuition to set up their displays need to understand how Intuition defaults work. The default values for the `LeftEdge`, `TopEdge`, `Width`, and `Height` of a screen can be a bit complex. This is partially due to the fact that they are implicitly specified by supplying a non-NULL `NewScreen` pointer to `OpenScreenTagList()`, and by the fact that while Intuition does define `STDSCREENWIDTH` and `STDSCREENHEIGHT` magic constants, there are no corresponding constants for standard left-edge and standard top-edge (which are likely non-zero!). The most foolproof strategy is to use a NULL `NewScreen` structure, using only tags to specify screen attributes, and omit all of `SA_Left`, `SA_Top`, `SA_Width`, and `SA_Height`. The correct defaults will come from the `SA_DClip`.

The other way (which is required if using a non-NULL `NewScreen`), is to supply the correct values for each component (either in the appropriate members of the `NewScreen` structure or with the appropriate tags). The correct values are:

```
left    = dclip.MinX
top     = dclip.MinY
width   = dclip.MaxX - dclip.MinX + 1
height  = dclip.MaxY - dclip.MinY + 1
```

Note that if your screen is shorter than the nominal height, other screens that are behind you can potentially be visible above your screen.

You are strongly encouraged to use Intuition screens instead of custom Views and ViewPorts. However, if you go that route, you can install a `DisplayClip` into your ViewPort's `ViewPortExtra` structure.

Screen Position and Sprites

User preference can shift the Intuition View origin, which in turn can affect the number of sprites available on your screen. If you use sprites other than the pointer sprite, you must take this into consideration, because a change in the upper-left corner of the Text Overscan rectangle shifts the entire coordinate system, so even carefully chosen fixed coordinates may get you into trouble. The upper-left corner of the `OSCAN_MAX` rectangle is the one which is at a fixed position with respect to the hardware fetch cycle (for a given mode). You can base a positional calculation on this if necessary.

There are some Display Clip positions for which the number of possible sprites varies depending on which fetch mode is used. The higher fetch modes have the advantage of reducing Chip RAM bus saturation, but you may prefer a lower bandwidth with more sprites. Graphics will drop the bandwidth to bring allocated sprites into view if `MakeVPort()` runs after you obtain your sprites via `GetExtSprite()`. In an Intuition environment, you should use `RemakeDisplay()` after you get your sprites (not call `MakeVPort()` directly).

▮ Avoiding The Workbench Screen

If you use Intuition screens in your titles, it is important to note an Intuition "feature" which can get in the way. Whenever the last screen opened in the system is closed, Intuition automatically opens up the Workbench screen. This can be quite cumbersome in titles which perform transitions between different screens.

Here is a short code sample for SAS/C that defeats this Intuition feature. Simply call `CloseScreenQuiet()` instead of calling `CloseScreen()` and all will be fine:

```
ULONG _asm OpenWorkBenchPatch(void)
{
    return(0);
}

VOID CloseScreenQuiet(struct CDUILib *CDUIBase, struct Screen *screen)
{
    APTR OWB;

    if (screen)
    {
        OWB = SetFunction(IntuitionBase, -0xd2, (APTR)OpenWorkBenchPatch);
        CloseScreen(screen);
        SetFunction(IntuitionBase, -0xd2, OWB);
    }
}
```

Note that the above code will only work starting with V39 Kickstart. Previous Kickstarts did not support this. Amiga CD³² has a V40 Kickstart.

▮ Localization

There are two ways software can be localized on Amiga CD³² systems. The `lowlevel.library`, discussed later in this chapter, offers a very simple mechanism that returns the number of the user's current preferred language. The second method involves using `locale.library` which provides a complete suite of localization services.

If your title does not use `locale.library`, it is best to remove the library from your CD. If your title does use `locale.library`, you must have a directory structure such as:

```
Locale (dir)
Catalogs (dir)
    français (dir)
        westchesterwarrior.catalog
    deutsch (dir)
        westchesterwarrior.catalog
    italiano (dir)
        westchesterwarrior.catalog
Languages (dir)
    français.language
    italiano.language
    deutsch.language
```

That is, you need a `Locale` directory which contains a `Catalogs` and a `Languages` subdirectory. There is no need to include the `Countries` directory that comes with the Workbench disk, this directory is only of use to the `Locale` preferences editor.

The Catalogs directory must contain one subdirectory for each language that your title supports. Within these directories, you can put the standard locale.library catalogs files (see locale.doc and catcomp.doc for more information).

The Languages directory must contain a language driver for each of the languages your title supports. These language drivers are supplied with Workbench 3.1 on the Locale disk. The LOCALE: assign must exist if you use locale.library. It should be made to the Locale directory described above.

■ New Amiga CD³² System Software

Five special system modules were created for the Amiga CD³² console:

- lowlevel.library
- nonvolatile.library
- freeanim.library
- cd.device
- CD-ROM file system

You can use these modules with full confidence that they will be upwardly compatible with future systems from Commodore. The lowlevel.library, nonvolatile.library and the CD-ROM file system will be part of Workbench 3.1 and will be shipped within the 3.1 Enhancer kits and with new machines that include Workbench 3.1. The freeanim.library will not be included with the normal Workbench distribution, but your code does not need to rely on its presence (see below for more on this). Finally, cd.device will be included with any Amiga CD³²-compatible solutions that Commodore will offer for its computer systems in future.

■ The nonvolatile.library

The purpose of this library is to allow an application to save small amounts of information to nonvolatile memory or a disk device without having to burden the application with determining which device to use. It provides a simple, common access method to whichever device the user chooses. This library also provides access to true nonvolatile memory present on the game machine. See the nonvolatile.library Autodocs in Chapter 4 for more information on this library.

■ The lowlevel.library

The lowlevel.library provides many functions useful for game developers. Among other things, this library provides the only means to read the Amiga CD³² game controller. In addition to the game controller support code, the lowlevel.library provides many functions often needed by game developers. Most of the functions in this library should not be used in the Amiga's normal multitasking environment—instead they are appropriate only for high-performance games which require the takeover of the system.

■ Compatibility with CDTV

Although the Amiga CD³² console can run old CDTV titles, you should never use any CDTV-specific system modules in your applications. There are known bugs and limitations in these modules when they are used in the Amiga CD³² environment. Future systems will not support these modules:

- cdtv.device
- playerprefs.library
- cardmark.device
- bookmark.device
- rmtm and CDTVPrefs

Game Programming Problems and Solutions

Speed of graphics rendering is the reason programmers usually cite for taking over the system. This is not a very good reason. In this section, some alternatives are described that you should consider instead of directly programming the hardware registers.

Blitter

If you have your own blitter routines that are faster than the routines in ROM, you can use them in a way that is friendly to the operating system. Use `OwnBlitter()` and `DisownBlitter()` to claim and relinquish ownership of the blitter registers.

First call `OwnBlitter()`, do any required setup, call `WaitBlit()`, poke the blitter registers, and then `DisownBlitter()` when all blits are done. This is a very low-overhead way to use direct hardware access; `OwnBlitter()` is only 3 instructions (counting RTS) when no one else is trying to use the blitter. You must use the graphics library `WaitBlit()`. This is as fast as possible, uses no CPU registers, and knows about blitter bugs. You cannot possibly write one that is more efficient and works on all Amigas.

For asynchronous blits, use the `QBlit()` and `QBSBlit()` functions. These have been rewritten for the 3.0 version of the operating system and now have quite low overhead. These let you use the blitter in an interrupt driven manner instead of polling for completion.

Copper

If you want to do full-screen animation, it is not necessary to take over the system at boot time or directly control the Copper registers. Instead, use a user-Copper list to cause a Copper interrupt on line 0 of your ViewPort.

The copper interrupt handler you install can signal a high-priority task that calls `ChangeVPBitMap()` (or `ScrollVPort()`) and `LoadRGB32()` to change both bitmap pointers and colors in sync. This allows for perfect 60 Hz animation even while moving the mouse as fast as possible and inserting floppy disks.

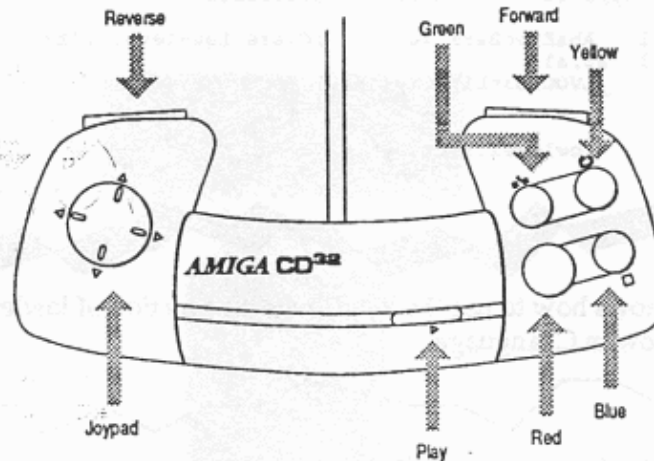
Under 3.0, you can also do this in an Intuition SA_Exclusive screen. You can tell if it was your screen which caused the copper interrupt by checking to see if the flag `VP_HIDE` is clear in your `ViewPort->Modes`.

If you really have to take over the Copper, first call `LoadView(NULL)`, do two `WaitTOF()`s, and then install your own copper lists in the `cop1/2jmp` registers. We do not recommend this though. Future chipsets will have faster and more efficient coppers with different registers. If you load copper registers directly, your code will break on these future systems.

```
temp=GfxBase->ActiView;
LoadView(NULL);
WaitTOF();
WaitTOF(); /* custom.copllc = ??? */
...
WaitTOF();
WaitTOF();
LoadView(temp);
custom.copllc=GfxBase->copinit;
```

Getting User Input

On the game machine, input will usually come from the "game controller." This is a hand-held controller similar to the type used on CDTV and the Nintendo SNES. It plugs into the standard, 9-pin D connectors on the side of the system. It has six action buttons, one start button and four directional arrows. Input could also come from a mouse or joystick.



All three types of game controllers can be conveniently managed with the `ReadJoyPort()` function in the `lowlevel.library`. This function is the only way to get Amiga CD³² game controller information. One drawback of this function is that it requires a polling loop. So applications designed to run in the Amiga's multitasking environment may want to use other methods to get joystick and mouse information. The mouse and joystick information can be obtained from `Intuition`, from the `gameport.device`, or from the hardware port registers directly.

Before looking at the hardware registers directly, it is imperative that you allocate the associated hardware resources using the `GPD_ASKCTYPE` and `GPD_SETCTYPE` commands of the `gameport.device`. You should also allocate the `POTGO` resource bits to acquire buttons 2 and 3. Code that demonstrates how to do this is listed on pages 89, 94-95 and 344 of the *Amiga ROM Kernel Reference Manual: Devices*, 3rd edition (ISBN 0-201-56775-X).

```

move.l   AbsExecBase,a6
lea      lowlib(pc),a1
move.l   #0,d0
jsr      LVOpenLibrary(a6)
tst.l    d0
beq.w    exit1
move.l   d0,a5
; Open low-level library
; version 0
; Make sure it opened
; Save LowLevelBase in A5

moveq.l   #1,d0
jsr      LVReadJoyPort(a6)
move.l    d0,d2
; Read status of game connector #1
; with the ReadJoyPort() function.
; Save joy 1 status in d2

andi.l    #JP_TYPE_MASK,d0
cmpi.l    #JP_TYPE_MOUSE,d0
bne.s     notmouse
; Get the controller type in d0
; Is it a mouse?

; It's a mouse.
notmouse
cmpi.l    #JP_TYPE_JOYSTK,d0
bne.s     notjoy
; It's not a mouse, so...
; Is it a joystick?

; It's a joystick

```



```

notjoy                                ;It's not a mouse or joystick, so...
    cmpi.l  #JP_TYPE_GAMECTLR,d0 ; Is it an AmigaCD32 game controller?
    bne.s   default

default                               ;It's an AmigaCD32 game controller
    ;It's type is unknown or not available

    move.l  _AbsExecBase,a6          ;Close Low-level Library
    move.l  a5,a1
    jsr     _LVOCloseLibrary(a6)

exitl
    rts
dc.b      'lowlevel.library',0
end

```

The fragment above shows how to use the ReadJoyPort() function of lowlevel.library. The same fragment is shown below in C language.

```

#define PORTNO 1L                    /* Controller connector to read */
struct Library *LowLevelBase; /* Global, define above main() */
...

ULONG status;
GameBase=OpenLibrary("lowlevel.library", 0L );
if(GameBase)
{
    /* Do set up, then, in main loop... */
    status = ReadJoyPort( PORTNO );
    switch( status & JP_TYPE_MASK )
    {
        case JP_TYPE_GAMECTLR: /* It's a game controller */
            break;
        case JP_TYPE_MOUSE:    /* It's a mouse */
            break;
        case JP_TYPE_JOYSTK:   /* It's a joystick */
            break;
        default:                /* Contoller type unknown or unavailable */
            break;
    }
    CloseLibrary(LowLevelBase);
}

```

Dealing with the Input Device

On the Amiga, the input.device is the system module that processes input events such as mouse movement and keystrokes passing them on to Intuition and other interested clients. If the system is not being completely shut down by use of the SCON_TakeOverSys tag to SystemControlA() the system overhead can be reduced by shutting down input.device. Obviously this should not be done if your application depends on input.device for gathering input.

There are two ways to do this. With lowlevel.library, the SystemControl() function can be used with the SCON_StopInput tag set to TRUE. This stops the input.device from using any CPU time and also prevents it from passing along input events to Intuition. This is the easiest way to shut down the input.device on the CD game machine.

A solution which is more multitasking-friendly is to install a high priority input handler which chokes off all events. To do this, you use the IND_ADDHANDLER and IND_REMHANDLER commands of

the input.device. Examples of how to use these are listed on pages 108 and 109 of the *Amiga ROM Kernel Reference Manual: Devices*, 3rd edition (ISBN 0-201-56775-X). By using a priority of 51, you can make your input handler first in the chain and prevent other handlers (like Intuition) from seeing input activity. This handler is also a convenient way to get keys and mouse events yourself. Simply store the raw keypresses and mouse moves in your own variables.

The CD game machine in its base configuration does not include a keyboard. However, a keyboard is planned as part of an expansion option for the system. Programs that want to support this option can get keyboard events in a variety of ways. The lowlevel.library functions AddKBInt() and RemKBInt() will add and remove code you provide to the system's keyboard interrupt servicing allowing you to detect raw keypresses.

If you prefer to poll the keyboard as part of your main processing loop, use the lowlevel.library function GetKey(). This returns a raw key value for the current key pressed and also any qualifier keys (such as Shift) that are pressed at the same time. To monitor the state of a given set of keys, you can use the lowlevel.library function QueryKeys(). You pass this function an array of raw key codes for the keys you want to know about and it returns their current state.

Another alternative that is multitasking friendly is to obtain input from the keyboard.device. This has the advantage of dealing with all the various keyboard timings, but it is easier by far to open an Intuition window and read either RAWKEY or VANILLAKEY IDCMP messages. These either give the raw key number pressed, or the character the key pressed represents (useful for international games).

Timing

For calculating elapsed time, there are a variety of possible solutions. For a variable-speed, high-precision, interrupt-driven time source, programs can use the timer functions of the lowlevel.library. These functions provide a way for your code to be called continuously at arbitrary intervals. For instance the example below shows how to wait for precisely 1000 microseconds.

```
struct Library *LowLevelBase;

struct TIData
{
    struct Task *atask;
    LONG        asignal;
};

/* This is just a structure */
/* to hold parameters passed */
/* to the interrupt routine. */

/* Here's the timer interrupt */
/* routine. It simply signals */
/* the parent task pointed to */
/* by register A1. */
VOID __asm __interrupt __saved
TIRoutine (register __a1 struct TIData *tidata)
{
    Signal( tidata->atask, 1L << tidata->asignal );
}

VOID
main ( int argc, char **argv)
{
    APTR          ti_handle;
    struct TIData ti_data;

    /* lowlevel.library timer handle */
    /* Pointer to interrupt arg.s */

    LowLevelBase = OpenLibrary ( "lowlevel.library", 0L);
    if (LowLevelBase)
    {
        ti_data.atask = FindTask( NULL );
        ti_data.asignal = AllocSignal(-1L);
        if (ti_data.asignal != -1)
        {
            /* Fill in TIData structure */
            /* so interrupt routine can */
            /* signal us. */
        }
    }
}
```

```

{
/* Attach the TIRoutine to the timer generated interrupt. */
ti_handle = AddTimerInt(TIRoutine, &ti_data);
if(ti_handle)
{
/* Start the timer; TIRoutine will be called every 1000 microsecs */
StartTimerInt(ti_handle, 1000L, TRUE);

Wait(1L << ti_data.asignal);      /* Wait for interrupt signal. */

StopTimerInt(ti_handle);          /* Tidy up. Free any system */
RemTimerInt(ti_handle);           /* resources used. */
}
FreeSignal(ti_data.asignal);
}
CloseLibrary(LowLevelBase);
}

```

Here is a similar example in 68020 assembly language. This time, the example flashes the display every 1024 ticks for a total delay of just over 10 seconds.

```

STRUCTURE TIData,0
    LONG asignal
    APTR atask
    LABEL TIData_sizeof
;

move.l    AbsExecBase,a6
lea.l     LowLib(pc),a1      ; Open low-level library
move.l     #0,d0             ; version 0
jsr       LVOOpenLibrary(a6)
tst.l     d0                 ; Make sure it opened
beq.w     exit3
move.l     d0,LowLevelBase    ; Save the base so we can close later

lea.l     IntLib(pc),a1      ; Open Intuition Library
move.l     #0,d0             ; version 0
jsr       LVOOpenLibrary(a6)
tst.l     d0                 ; Make sure it opened
beq.w     exit2
move.l     d0,IntuitionBase    ; Save the base so we can close later

move.l     #0,a1             ; Call Exec FindTask(NULL);
jsr       LVOFindTask(a6)     ; to get a pointer to the process.
lea.l     ti_data(pc),a0
move.l     d0,atask(a0)      ;

move.l     #-1,d0            ; Allocate a signal with Exec
jsr       LVOAllocSignal(a6)  ; AllocSignal(-1L)
cmpl.l     #-1,d0
beq.w     exit1
lea.l     ti_data(pc),a0      ; Store the allocated signal number
move.l     d0,asignal(a0)     ; in the asignal field of ti_data

move.l     LowLevelBase(pc),a6 ; Add in the timer interrupt
lea.l     TIRoutine(pc),a0
lea.l     ti_data(pc),a1
jsr       LVOAddTimerInt(a6)
tst.l     d0
beq.w     exit0
move.l     d0,ti_handle

move.l     d0,a1             ; Start the timer
move.l     #1000,d0
move.l     #TRUE,d1
jsr       _LVOSTartTimerInt(a6)

move.l     #10*1024,d4        ; Register d4 is the loop counter

```

```

mainloop
    move.l    _AbsExecBase,a6
    lea.l     ti_data(pc),a0
    move.l    asignal(a0),d1
    move.l    #1,d0
    asl.l     d1,d0
    jsr       _LVOWait(a6)

    move.l     d4,d0
    andi.l    #$000003ff,d0      ;Flash the screen every 1024 ticks
    bne.s     looptest

    move.l     IntuitionBase(pc),a6
    move.l     #0,a0
    jsr       _LVODisplayBeep(a6)

looptest
    subi.l    #1,d4
    tst.l     d4
    bne.w     mainloop

    move.l     LowLevelBase(pc),a6
    lea.l     ti_handle(pc),a1
    jsr       _LVOSTopTimerInt(a6)

    move.l     ti_handle(pc),a1
    jsr       _LVORemTimerInt(a6)

exit0
    move.l     _AbsExecBase,a6      ;Free the signal stored in the asignal
    lea.l     ti_data(pc),a1      ;field of ti_data
    move.l     asignal(a1),d0
    jsr       _LVOfreeSignal(a6)

exit1
    move.l     IntuitionBase(pc),a1 ;Close Intuition library
    jsr       _LVOCloseLibrary(a6)

exit2
    move.l     LowLevelBase(pc),a1 ;Close low-level library
    jsr       _LVOCloseLibrary(a6)

exit3
    rts

TIRoutine
    move.l     _AbsExecBase,a6      ;Prepare to call Exec Signal()
    move.l     asignal(a1),d1      ; register d0 contains the
    move.l     #1,d0               ; signal mask created from asignal
    asl.l     d1,d0                ;
    move.l     atask(a1),a1         ; register a1 points to the task
    jsr       _LVOSignal(a6)
    rts

;===== Data Area
LowLevelBase    dc.l    0
IntuitionBase   dc.l    0
loopcount       dc.l    0
ti_handle       dc.l    0
ti_data         ds.b     TIData_sizeof,0
                cnop 2
lowllib         dc.b     'lowlevel.library',0
                cnop 2
intulib         dc.b     'intuition.library',0
end

```

The lowlevel.library timer functions use the CIA chips (or equivalents) as the source for timing information. On systems prior to V40 that have no lowlevel.library, you will have to use the CIA timing facilities at a lower level. This involves a lot more housekeeping on the part of your application. For example code demonstrating how to do this, refer to pages 328-335 of the *Amiga ROM Kernel Reference Manual: Devices*, 3rd edition (ISBN 0-201-56775-X).

Another timing source is available through the timer.device called the E-Clock. The E-Clock is the clock used by the Motorola 680x0-family of processors to communicate with Motorola 8-bit devices. ReadEClock() is a timer.device function that provides a simple, low overhead way to determine

elapsed E-Clock time. It returns two values: a 64-bit timer value (the E-Clock count) and a 32-bit frequency value in ticks/second (the E-Clock frequency, related to the master frequency of the machine and different between PAL and NTSC systems). Usage of the E-Clock is shown on page 298 of the *Amiga ROM Kernel Reference Manual: Devices*, 3rd edition, (ISBN 0-201-56775-X). There is also a convenient interface to the EClock provided by the lowlevel.library function `ElapsedTime()`. See the Autodocs in Chapter 4 for more information.

For timings that require less granularity, the system vertical blank can be used as a special low-frequency timer. `WaitTOF()` will suspend your task until the top of frame is reached (and all vertical blank interrupt servers have been called). The frame rate depends on whether the system is running in a PAL or NTSC environment but is totally independent of the processor speed:

```
ULONG frate;
frate = (GfxBase->DisplayFlags & PAL) ? 50 : 60;
for( i=0; i < frate; i++ )
    WaitTOF();
```

As a convenience, V40 of the operating system includes a new field in `GfxBase` named `VBCounter`. The `VBCounter` field is a longword which is incremented by the system every vertical blank.

Audio

The audio device is one of the simplest, lowest overhead modules in the system. In general, there is no reason to hit the hardware directly. There are device commands to change the volume, period, frequency and data that is played on any of the four channels.

Some applications may want to modulate one voice with another or exceed the 28,000 byte/second sample playback rate by using CPU-driven audio instead of DMA-driven audio. In that case, direct

```
VOID main( int argc, char **argv)
{
    struct IOAudio *myAIReq=NULL;
    struct MsgPort *myAIReply=NULL;
    UBYTE chans[] = {15}; /* Allocate all 4 audio channels */
    BYTE dev = -1;

    myAIReq=(struct IOAudio *)AllocMem(sizeof(struct IOAudio),MEMF_PUBLIC );
    if(myAIReq)
    {
        myAIReply=CreatePort(0,0);
        if(myAIReply)
        {
            myAIReq->ioa_Request.io_Message.mn_ReplyPort = myAIReply;
            myAIReq->ioa_Request.io_Message.mn_Node.ln_Pri = 127;
            myAIReq->ioa_Request.io_Command = ADCMD_ALLOCATE;
            myAIReq->ioa_AllocKey = 0;
            myAIReq->ioa_Data = chans;
            myAIReq->ioa_Length = sizeof(chans);

            dev=OpenDevice("audio.device",0L,(struct IORequest *)myAIReq,0L);
            if(! dev)
            {
                /* Your code that uses audio hardware registers goes here */

                CloseDevice(myAIReq);
            }
            else { /* Couldn't get all 4 channels. */ }
            DeletePort( myAIReply );
        }
        FreeMem( myAIReq, sizeof(struct IOAudio) );
    }
}
```



```

;Imports (amiga.lib) =====
xref _CreatePort
xref _DeletePort
;Exports=====
xdef _SysBase
;Equates=====
SysBase EQU 4
SIZEOFCHANS EQU 1
; channel allocation array

move.l SysBase,a6 ;Allocate memory for the IOAudio
move.l #ioa_SIZEOF,d0
move.l #MEMF_PUBLIC,d1
jsr _LVOAllocMem(a6)
move.l d0,a3 ;a3 = a pointer to my IOAudio
tst.l d0 ; request block
beq exit3

move.l #0,-(sp) ;Call the amiga.lib C function
move.l #0,-(sp) ; CreatePort() to create a
jsr _CreatePort ; reply port
addq.l #8,sp
move.l d0,a4 ;a4 = a pointer to my IOAudio
tst.l d0 ; reply message port
beq exit2

;Initialize the IOAudio request
myAIReq
move.w #0,ioa_AllocKey(a3) ; ->ioa_AllocKey
lea.l chans(pc),a1 ;
move.l a1,ioa_Data(a3) ; ->ioa_Data
move.l #SIZEOFCHANS,ioa_Length(a3) ; ->ioa_Length
move.w #ADCMD_ALLOCATE,ioa_Command(a3) ; ->ioa_Request.io_Command
move.l a4,MN_REPLYPORT(a3) ; ->ioa_Request.io_Message.
; mn_ReplyPort
; ->ioa_Request.io_Message.
; mn_Node.ln_Pri

move.b #127,LN_PRI(a3)

lea.l adname(pc),a0
moveq.l #0,d0 ;OK now call open device, unit 0,
move.l a3,a1 ; on the audio.device
moveq.l #0,d1
jsr _LVOpenDevice(a6)
tst.l d0 ; Make sure it opened
bne.w exit1

move.l #SIGBREAKF_CTRL_C,d0 ;Wait() for Ctrl-C
jsr _LVOWait(a6)

move.l a3,a1 ;Call CloseDevice() to free
jsr _LVOCloseDevice(a6) ; channels
exit1
move.l a4,-(sp) ;Delete reply port
jsr _DeletePort
addq.l #4,sp
exit2
move.l a3,a1 ;Free the IOAudio memory
move.l #ioa_SIZEOF,d0
jsr _LVOfreeMem(a6)
exit3
rts

; Data Area =====
chans dc.b 15
cnop 2
adname dc.b 'audio.device',0
end

```

hardware access is required. If you must hit the audio hardware, you can ask for the channel you need with the highest priority (127). The code above shows how:

The audio channels will never be stolen from you, nor will they be used by anyone else until you give the channels back to the system.

► Generating Random Numbers

In all games you will need at least some element of randomness in the game to prevent the user from feeling bored during repeat plays. There are a variety of ways to generate random numbers.

The code below is based on the random-number routine from *Semi-Numerical Algorithms: The Art of Computer Programming, Vol. 2*, by Donald Knuth. The function `rand_word()` will return a random 16-bit value and has an extremely long period. The `rand_word()` function uses a table for speed; the table is initialized with another random number generator (which is slightly slower because it has a multiply in it).

```
init table:
; initialize random # generator - pass d0 = a random seed (based off of time)
; trashes d0/d1,a0
    moveq    #54,d1
    lea      rand_table(a6),a0
1$:    move   d0,(a0)+
    mulu     #5lef,d0
    add      #5dff,d0
    dbra     d1,1$
    move.l   a0,(a0)
    move.l   #rand_table+24*2,j_index(a6)
    rts

rand word:
; return a random word in d0
; trashes a0/a1
    lea      rand_table(a6),a1
    move.l   j_index(a6),a0
    move.w   -(a0),d0
    cmp.l    a0,a1
    bne.s    2$
    lea      rand_table+55*2(a6),a0
2$:    move.l   a0,j_index(a6)
    move.l   k_index(a6),a0
    add      -(a0),d0
    move     d0,(a0)
    cmp.l    a0,a1
    bne.s    3$
    lea      rand_table+55*2(a6),a0
3$:    move.l   a0,k_index(a6)
    rts

    section variables

;!! rand_table must precede k_index
rand_table    ds.w    55
k_index       dc.l    0
;!! k_index must follow rand_table

j_index       dc.l    0

    end
```

You will want to use a different seed value each time your program is run. The best way to do this is by using the `ReadEClock()` routine of the timer device to obtain a seed value. The `EClock` is a high speed, free-running, 64-bit timer that increments every 279.5 ns. This is fast enough so that now matter how the the program is started, you are very likely to get a different reading from the `EClock` from one game to the next. (See the earlier section on Timing for more information.)

An alternative to the table-driven random number generator above is listed in the `Random()` and `RandomSeed()` assembly-language functions below. These functions are callable from either C or

assembly language. The `amiga.lib` link library provided by Commodore has additional random number routines `FastRand()` and `RangeRand()` that can be called only from a C-language program. Here's a C program that demonstrates how to call these functions to obtain a random number.

```

/* test_random.c - Execute me to compile me with SAS C 5.10
LC -bl -cfist -v -j73 test_random.c
Blink FROM LIB:c.o,test_random.o,random.o TO test_random LIBRARY
LIB:LC.lib,LIB:Amiga.lib
quit

* Shows the use of both the Amiga.lib RangeRand() function and the
* random.asm Random() function it is linked with to demonstrate
* two ways for generating random numbers.
*
* Uses Intuition CurrentTime() to get a seed for the random number generator
* Alternately, you could use the 2.0 timer.device function ReadEClock
*/

#include <exec/types.h>
#include <exec/memory.h>
#include <dos/dos.h>
#include <intuition/intuition.h>

#include <clib/exec_protos.h>
#include <clib/dos_protos.h>
#include <clib/intuition_protos.h>
#include <clib/graphics_protos.h>

#include <stdlib.h>
#include <stdio.h>
#include <string.h>

#ifdef LATTICE
int CXBRK(void) { return(0); } /* Disable Lattice CTRL/C handling */
void chkabort(void) { return; } /* really */
#endif

#define MINARGS 1

UBYTE *vers = "\0$VER: test_random 39.1";

/* Option 1: Random.asm RandomSeed() and Random() and functions
*
* RandomSeed() - Seed with a variable number such as the current time
*
* Random() - Pass limit of 0-65535, returns a number between 0 and (limit-1)
*/
extern void RandomSeed(ULONG seedvalue);
extern int Random(LONG limit);

/* Option 2:
* Amiga.lib ULONG RangeSeed and RangeRand() function
*
* RangeRand() - Pass limit of 0-65535, returns a number between 0 and (limit-1)
*/
#include <clib/alib_protos.h>
extern ULONG far RangeSeed;

void testrand(LONG limit);

struct Library *IntuitionBase;

void main(int argc, char **argv)
{
    ULONG secs,mics,seed;

    if(IntuitionBase = OpenLibrary("intuition.library",0))
    {
        /* We'll use CurrentTime() to get an initial "seed" value.
        * You could also use the 2.0 timer.device function ReadEClock.
        * If you use the same seed value, you'll get the same
        * sequence of "random" values each time you run the program.
        */
    }
}

```

```

CurrentTime(&secs,&mics); /* Use as our seed */
seed = secs + mics;
printf("secs=%ld mics=%ld, seed = $08lx\n",secs, mics, seed);

/* We'll be testing both the random.asm Random() function and the
 * amiga.lib RangeRand() function so we need to seed each with a
 * starter random value by using the seed method each requires
 */
RandomSeed(seed); /* seed random.asm Random() */
RangeSeed = seed; /* seed Amiga.lib RangeRand() */

/* Call both functions to get some random numbers, passing limit
 * Both functions return random numbers from 0 through limit-1
 * Note: however that 0 cannot be passed as limit to RangeRand()
 */
testrand(3);
testrand(11);
testrand(512);
testrand(65535);

CloseLibrary(IntuitionBase);
}

```

```
void testrand(LONG limit)
```

```

{
    int k, ktries;

    ktries = 10;

```

```

    /* Note: Can pass limit 0-65535, returns number from 0 to limit-1 */
    printf("Random(): %ld random values between 0 and %ld:\n",ktries,limit-1);
    for(k=0; k<ktries; k++) printf("%ld ",Random(limit));
    printf("\n");

```

```

    /* Note: Can pass limit 1-65535, returns number from 0 to limit-1 */
    printf("RangeRand(): %ld random values between 0 and %ld:\n",ktries,limit-1);
    for(k=0; k<ktries; k++) printf("%ld ",RangeRand(limit));
    printf("\n\n");
}

```

* Random number generator

XDEF	RandomSeed,Random	assembler entry points
XDEF	_RandomSeed,_Random	C entry points
CODE		

```

;=====
; NAME
;   RandomSeed - seed random number generator, call once at beginning of
;               your program. CurrentTime provides useful values for this
;
; SYNOPSIS
;   RandomSeed( SeedValue )
;               D0
;
; FUNCTION
;   Seeds the random number generator
;
; INPUTS
;   SeedValue - a longword containing any value you like
;
; RESULT
;   Random number generator is initialised
;
; BUGS/LIMITATIONS
;   None that I know of
;=====

```

```

RandomSeed    MOVE.L 4(SP),D0      entry point for C functions
RandomSeed    ADD.L D0,D1          user seed in d0 (d1 too)
              MOVEM.L D0/D1,RND

; drops through to the main random function (not user callable)

LongRnd        MOVEM.L D2-D3,-(SP)
              MOVEM.L RND,D0/D1    D0=LSB's, D1=MSB's of random number
              ANDI.B #$0E,D0       ensure upper 59 bits are an...
              ORI.B  #$20,D0       ...odd binary number
              MOVE.L D0,D2
              MOVE.L D1,D3
              ADD.L D2,D2           accounts for 1 of 17 left shifts
              ADDX.L D3,D3         [D2/D3] = RND*2
              ADD.L D2,D0
              ADDX.L D3,D1         [D0/D1] = RND*3
              SWAP  D3             shift [D2/D3] additional 16 times
              SWAP  D2
              MOVE.W D2,D3
              CLR.W D2
              ADD.L D2,D0          add to [D0/D1]
              ADDX.L D3,D1
              MOVEM.L D0/D1,RND    save for next time through
              MOVE.L D1,D0         most random part to D0
              MOVEM.L (SP)+,D2-D3
              RTS

;=====
; NAME
; Random - returns a random integer in the specified range
;
; SYNOPSIS
; RndNum = Random( UpperLimit )
; D0
; D0
;
; FUNCTION
; returns a random integer in the range 0 to UpperLimit-1
;
; INPUTS
; UpperLimit - a long(or short will do) in the range 1-65535
;
; RESULT
; a random integer is returned to you, real quick!
;
; BUGS/LIMITATIONS
; range was limited to 0-65535 to avoid problems with the DIVU instruction
; which can return real wierd values if the result is larger than 16 bits.
;=====

Random        MOVE.L 4(SP),D0      C entry point
Random        MOVE.W D2,-(SP)
              MOVE.W D0,D2         save upper limit
              BEQ.S 10$            range of 0 returns 0 always
              BSR.S LongRnd        get a longword random number
              CLR.W D0             use upper word (it's most random)
              SWAP D0
              DIVU.W D2,D0         divide by range...
              CLR.W D0            ...and use remainder for the value
              SWAP D0             result in D0.W
10$           MOVE.W (SP)+,D2
              RTS

RND           BSS
              DS.L 2              random number
              END

```



```

RandomSeed    MOVE.L 4(SP),D0      entry point for C functions
RandomSeed    ADD.L D0,D1          user seed in d0 (d1 too)
RandomSeed    MOVEM.L D0/D1,RND

; drops through to the main random function (not user callable)

LongRnd        MOVEM.L D2-D3,-(SP)
LongRnd        MOVEM.L RND,D0/D1    D0=LSB's, D1=MSB's of random number
LongRnd        ANDI.B #$0E,D0      ensure upper 59 bits are an...
LongRnd        ORI.B  #$20,D0      ...odd binary number
LongRnd        MOVE.L D0,D2
LongRnd        MOVE.L D1,D3
LongRnd        ADD.L D2,D2          accounts for 1 of 17 left shifts
LongRnd        ADDX.L D3,D3        [D2/D3] = RND*2
LongRnd        ADD.L D2,D0
LongRnd        ADDX.L D3,D1        [D0/D1] = RND*3
LongRnd        SWAP D3            shift [D2/D3] additional 16 times
LongRnd        SWAP D2
LongRnd        MOVE.W D2,D3
LongRnd        CLR.W D2
LongRnd        ADD.L D2,D0          add to [D0/D1]
LongRnd        ADDX.L D3,D1
LongRnd        MOVEM.L D0/D1,RND    save for next time through
LongRnd        MOVE.L D1,D0        most random part to D0
LongRnd        MOVEM.L (SP)+,D2-D3
LongRnd        RTS

```

```

;=====
; NAME
; Random - returns a random integer in the specified range
;
; SYNOPSIS
; RndNum = Random( UpperLimit )
; D0      D0
;
; FUNCTION
; returns a random integer in the range 0 to UpperLimit-1
;
; INPUTS
; UpperLimit - a long(or short will do) in the range 1-65535
;
; RESULT
; a random integer is returned to you, real quick!
;
; BUGS/LIMITATIONS
; range was limited to 0-65535 to avoid problems with the DIVU instruction
; which can return real wierd values if the result is larger than 16 bits.
;=====

```

```

Random        MOVE.L 4(SP),D0      C entry point
Random        MOVE.W D2,-(SP)
Random        MOVE.W D0,D2          save upper limit
Random        BEQ.S 10$,            range of 0 returns 0 always
Random        BSR.S LongRnd         get a longword random number
Random        CLR.W D0              use upper word (it's most random)
Random        SWAP D0
Random        DIVU.W D2,D0          divide by range...
Random        CLR.W D0              ...and use remainder for the value
Random        SWAP D0              result in D0.W
10$           MOVE.W (SP)+,D2
10$           RTS

RND           BSS
RND           DS.L 2               random number
RND           END

```

Double Speed and Other CD Drive Issues

The Amiga CD³² has an added feature that allows the CD-ROM drive to perform in a double-speed mode of operation (300Kb/s, 150 frames/s). To have the drive operate in double-speed mode, you need to do two things. The first thing is very important. You need to put your data as close to the outermost tracks as possible. Double-speed works best when your data is on the outer tracks. The ISOCD utility program on the ISO Tools Disk gives you the ability to pad the beginning of the disk allowing you to place your data as far out as possible. The drive will not prohibit you from using double-speed on the inner tracks of the disk but you should use double-speed only on the outer tracks for best results.

The second thing you need to do is tell the drive to do double-speed. This is done by issuing the CD_CONFIG command of the cd.device as shown in the example below. Depending on the tags you use with the CD_CONFIG command, you can double the speed of either normal reads or CDXL-style reads. Usually, you will only want to configure normal reads for double-speed.

The MaxReadSpeed() function below shows how to configure normal reads (cd.device/CD_READ) to the maximum speed the drive is capable of. The SetReadXLSpeed() below shows how to configure CDXL-style reads (cd.device/CD_READXL) to a desired speed. If the only thing you want to use double-speed for is a CDXL sequence, then you only need to use the SetReadXLSpeed() function. You do not need to configure the CD_READ command to do double-speed as well. Just make sure that you XL sequence is on the outer tracks of the disk.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include <exec/io.h>
#include <exec/memory.h>
#include <utility/tagitem.h>
#include <devices/cd.h>

#include <clib/exec_protos.h>
#include <pragmas/exec_pragmas.h>

extern struct SysBase *SysBase;

struct IOStdReq *IOR;
struct MsgPort *Port;

struct CDInfo CDInfo;
struct TagItem ConfigList[] =
{ 0, 0 },
{ TAG_END, 0 }
};

/*****
 * int MaxReadSpeed(void) -- Set the speed of the READ command to the
 *                          maximum that the drive can handle.
 *****/

int MaxReadSpeed(void) {
    int success = 0;

    if (Port = CreateMsgPort()) {
        if (IOR = CreateIORequest(Port, sizeof(struct IOStdReq))) {
            if (!OpenDevice("cd.device", 0L, IOR, 0L)) {
                IOR->io_Command = CD_INFO;
                IOR->io_Data = (APTR)&CDInfo;
                IOR->io_Offset = 0;
                IOR->io_Length = sizeof(struct CDInfo);
                DoIO(IOR);
            }
        }
    }
}
```

```

        if (!IOR->io_Error) { /* Success? */

            /* Modify TagList */
            ConfigList[0].ti_Tag = TAGCD_READSPEED;
            ConfigList[0].ti_Data = CDInfo.MaxSpeed;

            /* Configure drive to maximum speed */
            IOR->io_Command = CD_CONFIG;
            IOR->io_Data = (APTR)&ConfigList;
            IOR->io_Length = 0;
            DoIO(IOR);

            /* Report success */
            if (!IOR->io_Error) success = 1;

            CloseDevice(IOR); /* Close cd.device */
        }
        DeleteIORequest(IOR); /* Delete IORequest */
        DeleteMsgPort(Port); /* Delete reply port */
    }
    return(success); /* Return success */
}

/*****
 * int SetReadSpeed(int Speed) --- Set the speed of the READXL command to
 *                               the desired frame rate.
 *****/

int SetReadXLSpeed(int Speed) {
    int success = 0;

    if (Port = CreateMsgPort()) { /* Create reply port */
        if (IOR = CreateIORequest(Port, sizeof(struct IOStdReq))) { /* Create IORequest */
            if (!OpenDevice("cd.device", 0L, IOR, 0L)) { /* Open cd.device */

                /* Modify TagList */
                ConfigList[0].ti_Tag = TAGCD_READXLSPEED;
                ConfigList[0].ti_Data = Speed;

                /* Configure drive to desired speed */
                IOR->io_Command = CD_CONFIG;
                IOR->io_Data = (APTR)&ConfigList;
                IOR->io_Length = 0;
                DoIO(IOR);

                if (!IOR->io_Error) success = 1; /* Report success */

                CloseDevice(IOR); /* Close cd.device */
                DeleteIORequest(IOR); /* Delete IORequest */
                DeleteMsgPort(Port); /* Delete reply port */
            }
            return(success); /* Return success */
        }
    }

    /*****
     * Try setting the READ speed to the maximum the drive can handle and try
     * setting the speed of the READXL command to 300Kb/s.
     *****/

    void main (int argc, char **argv) {

        /* Set the READ commands to maximum speed */
        if (MaxReadSpeed()) printf("Read commands are at maximum speed");

        /* Try to set READXL frame rate to 150 frames/second (300Kb/s) */
        if (!SetReadXLSpeed(150)) printf("Drive can't support double-speed");
    }
}

```

Other CD Drive Issues

The unit name for the CD-ROM drive on the game machine is CD0:. This is the same name used on CDTV and on the A570 CD-ROM peripheral for the A500. However keep in mind that other vendors who provide CD-ROM drives for the Amiga may use a different drive name. For future compatibility try to use paths relative to your application's volume name rather than relative to the unit name. The system also provides a special process-relative assign named PROGDIR:. This assignment exists for every process except those on the resident list and will be the directory from which the process's executable was loaded.

Another drive consideration is that titles that could run on an A4000 or A1200 equipped with a CD-ROM drive will not be able to boot from CD on such systems (no ISO filesystem in ROM in V39 and earlier). This is one excellent reason to enable your product to run from the Workbench.

Similarly, products that use the CD-ROM at the lowest level, i.e., by making calls to cd.device, are unlikely to work with CD-ROM drives from other vendors whose implementations of the ISO filesystem is different. To avoid this compatibility pitfall, stick to the higher-level filesystem interface to the CD drive. Just use the dos.library.

Memory

The CD game machine will have two megabytes of Chip RAM in its base configuration. There are a fairly large number of wait-states when accessing Chip RAM. ROM is zero wait-states. Due to the slow RAM speed, it may be better to use calculations for some things that you might have used tables for on the A500. Add-on RAM (Fast RAM) will probably be faster than Chip RAM, so it is worth segmenting your game so that parts of it can go into Fast RAM if available.

CPU Speed Issues

With the 68EC020 processor used in all CD game machines, there are some special coding techniques you can use to improve performance. For example, it is critical that you code any important loops to execute entirely from the on-chip 256-byte cache. A straight line loop 258 bytes long will execute far slower than a 254 byte one. The '020 is a 32-bit chip. Longword accesses will be twice as fast when they are aligned on a longword boundary. Aligning the entry points of routines on 32-bit boundaries can help, also. You should also make sure that the stack is always longword aligned. Write accesses to Chip RAM incur wait-states. However, other processor instructions can execute while results are being written to memory:

```
move.l d0, (a0)+ ; store x coordinate
move.l d1, (a0)+ ; store y coordinate
add.l d2, d0 ; x+=deltax
add.l d3, d1 ; y+=deltay
```

will be slower than:

```
move.l d0, (a0)+ ; store x coordinate
add.l d2, d0 ; x+=deltax
move.l d1, (a0)+ ; store y coordinate
add.l d3, d1 ; y+=deltay
```

The 68020 adds a number of enhancements to the 68000 architecture, including new addressing modes and instructions. Some of these are unconditional speedups, while others only sometimes help.

■ New Addressing Modes

Scaled Indexing. The 68000 addressing mode (disp,An,Dn) can have a scale factor of 2, 4 or 8 applied to the data register on the 68020. This is totally free in terms of instruction length and execution time. For example:

```

68000      68020
add.w      d0,d0      move.w      (0,a1,d0.w*2),d1
move.w      (0,a1,d0.w),d1

```

16-bit Offsets on An+Rn Modes. The 68000 only supported 8-bit displacements when using the sum of an address register and another register as a memory address. The 68020 supports 16-bit displacements. This costs one extra cycle when the instruction is not in cache, but is free if the instruction is in cache. 32-bit displacements can also be used, but they cost 4 additional clock cycles.

Data Registers Used as Addresses. Register (d0) is 3 cycles slower than (a0), and it only takes 2 cycles to move a data register to an address register, but this can help in situations where there is no free address register.

Memory Indirect Addressing. These instructions can help in some circumstances when there are not any free registers to load a pointer into. Otherwise, they lose.

■ New Instructions

Extended Precision Divide and Multiply. The 68EC020 can perform 32x32->32, 32x32->64 multiplication and 32/32 and 64/32 division. These are significantly faster than the multi-precision operations which are required on the 68000.

EXTB. The sign extend byte to longword instruction is faster than the equivalent EXT.W EXT.L sequence on the 68000.

CMPI and TST. Compare immediate and TST work in program-counter relative mode on the 68020.

Shift Instructions. On the '020, all shift instructions execute in the same amount of time, regardless of how many bits are shifted. Note that ASL and ASR are slower than LSL and LSR. The break-even point on ADD Dn,Dn versus LSL is at two shifts.

Bit Field Instructions. BFINS inserts a bitfield, and is faster than 2 MOVES plus an AND and an OR. This instruction can be used nicely in fill routines or text plotting. BFEXTU/BFEXTS can extract and optionally sign-extend a bitfield on an arbitrary boundary. BFFFO can find the highest order bit set in a field. BFSET, BFCHG, and BFCLR can set, complement, or clear up to 32 bits at arbitrary boundaries.

■ General Do's and Don'ts

- Do clear unused bits when writing, and mask out unused or unneeded bits when reading.
- Don't use timing loops. The reasons should be obvious.
- Don't write self-modifying code unless you know how instruction caches work.
- If you are hardware banging, don't assume anything about the initial contents of the display registers (or any other registers) when your program is started. Initialize everything.
- If using ViewPorts, be sure to have a properly allocated ViewPortExtra. Some graphics calls are faster when one is present.
- Do note well the warning around the copinit structure.